National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

ALGORITHMS FOR THE SPARE CAPACITY DESIGN OF MESH RESTORABLE NETWORKS

BY

BRADLEY DAVID VENABLES ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF ELECTRICAL ENGINEERING

Edmonton, Alberta
Fall, 1992

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-77143-7

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR:	**Bradley David Venables**
TITLE OF THESIS:	**Algorithms for the Spare Capacity Design of Mesh Restorable Networks**
DEGREE:	**Master of Science**
YEAR THIS DEGREE GRANTED:	**1992**

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material whatever without the author's prior written permission.

B Venables

7311-156 Street
Edmonton, Alberta
CANADA, T5R 1X4

August 18, 1992

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **ALGORITHMS FOR THE SPARE CAPACITY DESIGN OF MESH RESTORABLE NETWORKS** submitted by **Bradley David Venables** in partial fulfillment of the requirements for the degree of **Master of Science**.

W.D. Grover

J. J. Harms

W.B. Joerg

August 18, 1992

I dedicate this thesis to my wife, my parents, and especially the memory of my sister, who prevailed in spite of my ambitiousness.

Abstract

Future systems for planning and operating mesh-restorable networks will require an efficient algorithm for spare capacity placement that assures restorability with a minimum of total protection capacity. This thesis proves that the problem of optimal spare capacity placement (SCP) in a mesh-restorable network is NP-hard. From there, this work considers two heuristic strategies to solve the SCP problem in a near-optimal way within reasonable time constraints.

The Spare Link Placement Algorithm (SLPA) establishes an SCP by employing iterative spare link addition. SLPA selects each addition to produce the greatest incremental change in network restorability. Here, it is proven, theoretically and experimentally, that SLPA is strictly polynomial in time complexity. The Iterated Cutsets Heuristic (ICH) formulates SCP as a Linear Programming (LP) problem subject to constraints based on a subset of cutset flows of the network. Iteration and heuristic rules are used to develop the constraint set required by ICH for an SCP solution. It is proven, theoretically and experimentally, that ICH has exponential worst-case time complexity.

Experimental tests of SLPA and ICH are performed on 36 trial networks. The two approaches are compared for use as near-optimal SCP tools in initial design and subsequent maintenance of a continually growing network. Network providers can select a method for survivable network design and management systems from this comparison data.

Acknowledgment

I thank Dr. W. G. Grover, Director, Networks & Systems - TR Labs, for supervising this research project. I also thank Dr. Grover for establishing my current career path six years ago when he first enticed a redirection to the field of engineering research. I thank Mr. T. Bilodeau for many of the original ideas within the SLPA implementation described and tested here. I especially thank Dr. M. MacGregor for technical, moral, and computing support. Dr. MacGregor was both a co-supervisor and a member of the supervisory committee without any formal academic acknowledgment of those roles.

I also acknowledge the following contributions to my education:

TR Labs, NSERC and the University of Alberta for financial support.

TR Labs, Mike MacGregor, and Bob Gregorish for computing environment support.

Rohit Sharma, Richard Tse, Ken Benterud, Tom Bilodeau, Yvonne DenOtter, Tony Dunn and the rest of the staff and students at TR Labs for providing a nourishing environment for growth.

Jim Slevinsky for co-writing Appendix C and general attempts not to consume too much of my time as the eleventh hour approached.

Dr. Harms and Dr. Joerg for valuable input as members of the supervisory committee.

Dr. Conradi for exposing me to the potential of a graduate school education.

Tom Willekes for sharing the weight of undergraduate engineering.

Rohit Sharma for sharing with me the Meaning of Liff Dictionary for true technical writing.

LaVonne for general support.

In closing, I thank TRLabs for the tunnel and BNR for the light at the end.

Table of Contents

1 Introduction 1
1.1 Outline 2

2 Mesh Transport Networks 4
2.1 The Network 4
2.2 Restoration Re-configuration 4
2.3 Restoration as a Routing Problem 5
2.4 Topology Design and Working Path Routing 6
2.5 Path or Span Restoration 8
2.6 Mesh Versus Ring Restoration 10

3 Performance Criteria and Experiments 14
3.1 Computational Complexity (Time and Space) 14
3.2 Restorability 14
3.3 Total Network Capacity 15
3.4 Ease of Implementation 15
3.5 Accommodation of Network Growth 16
3.6 Over-Restorability 16
3.7 Experimental Network Models 17
3.7.1 Network Topology Characteristics 17
3.7.2 Generation of Test Networks 17
3.7.3 The Networks 19

4 SCP Problem Formulation 20
4.1 Introduction to Graph Theory 20
4.1.1 Cuts and Cutsets 21
4.1.2 Efficiency Analysis 21
4.1.2.1 Why Asymptotic Analysis? 21
4.1.2.2 Parameters of Complexity 23
4.1.2.3 Average-case Analysis 23
4.1.3 Quasi-Planarity 24
4.1.4 Locality Information 24
4.1.5 Common Data Structures 27
4.1.5.1 Adjacency Matrices and Lists 27
4.1.5.2 Binary Heaps 29
4.1.6 Common Procedures 31
4.1.6.1 Depth First Search (DFS) 32
4.1.6.2 A Fast Implementation of Dijkstra's Algorithm 33
4.2 Network Flows 39
4.3 Formulation of the Optimal Spare Capacity Placement Problem 40

5 The Complexity of Spare Capacity Placement 43
5.1 Introduction to Complexity Theory 43
5.2 Complexity Proof of Spare Capacity Placement 44
5.2.1 Reduction of a Hamiltonian Cycle to a Spare Capacity Placement in a Network With One Working Link on Each Span 45
5.2.2 Conversions of the Input and Output Data 46
5.2.3 Conclusion 47
5.3 Complexity of the Direct Approach to Optimum SCP 47
5.3.1 Number of Cutsets in Direct Cutsets Algorithm 47
5.3.2 Identifying Cutsets in Direct Cutsets Algorithm (DCA) 48
5.3.3 Methods of Generating Subsets of DCA Constraints 48
5.3.4 Application of Subsets of Constraints to a Heuristic Implementation 52

6 Iterative Cutsets Heuristic 53
6.1 Introduction 53
6.2 ICH With Restoration Path Limit (RPL) 57
6.3 ICH With Transmission System Modularity 58
6.4 The Theoretical Complexity of ICH 59
6.4.1 Worst-Case Complexity 59
6.4.2 Average-Case Complexity 60
6.5 Linear Programming Versus Integer Programming 63
6.6 Restoration Type and Restorability 65
6.7 ICH Summary 66

7 The Spare Link Placement Algorithm 68
7.1 Introduction to SLPA 68
7.1.1 Initialization 70
7.1.2 FS Phase 70
7.1.3 DT Phase 74
7.1.4 Variants in SLPA Implementation 77
7.2 Implementation of SLPA 77
7.2.1 Span Locality 77
7.2.2 Effect of Span Operations 78
7.2.3 Restorability Calculations based on metaDijkstra() 79
7.2.3.1 Span Localities with metaDijkstra() 79
7.2.3.2 The dijk_calc_rest() Procedure 80
7.2.4 Restorability Calculations based on a Path-Table 81
7.2.4.1 The Structure of the Path-Table 82
7.2.4.2 Evaluating Rs,i With the Path-Table 83
7.2.4.3 Evaluating R_n With the Path-Table 85
7.2.4.4 Computational Complexity Associated with the Path-Table 87

7.2.5 Implementation of the SLPA Modules Using Locality Information 88
7.2.5.1 Add One Spare Link 88
7.2.5.2 Add Two Spare Links Simultaneously 89
7.2.5.3 Delete One Spare Link 90
7.2.5.4 Add One Spare, While Deleting Two Spares 90
7.2.5.5 Add Two Spare Links While Deleting Three Spare Links 94
7.3 Complexity Analysis of SLPA 96
7.3.1 Worst Case Analysis 97
7.3.2 Experimental Results 98
7.4 Restoration Type and Restorability 103
7.5 Transmission System Modularity 105
7.6 Summary 106

8 Comparison of SCP Heuristics 107
8.1 Execution Time 107
8.2 Restorability 110
8.3 Capacity Efficiency 111
8.4 Accommodation of Network Growth 102
8.6 Over-Restorability 104
8.7 Ease of Implementation 105
8.8 Discussion and Recommendations 106

9 Joint Provisioning and Restorability 119
9.1 Introduction 119
9.2 Joint Provisioning and Restorability Strategies 120
9.3 Experiments 121
9.4 Network Capacity Changes 122
9.5 Redundancy Levels 122
9.6 Summary of Joint Provisioning and Restorability 123

10.0 Summary 125
10.1 Further Research 127
10.1.1 Max-flow Strictly Constrained by RPL 127
10.1.2 SCP Synthesis Algorithm With Simulated Annealing 127
10.1.3 SLPA With Span Specific RPL 128
10.1.4 Usable Algorithm for Polynomial Time-Complexity Linear Program 128
10.1.5 Exact Locality Identification in Polynomial Time 128
10.1.6 Network Topology Design in Which k-shortest Path Flow Equals Max-flow 128
10.1.7 Fast Simulation of Known Restoration Algorithms for Use in SLPA for Checking Restorability 129

Bibliography 130
Appendix A Known Telecommunications Network Topologies 135
Appendix B Experimental Random Study Network Topologies 140
Appendix C ICH and SLPA Software User Documentation 150

List of Tables

3.1 Study Network Parameters 19

4.1 Terminology Comparison of Telecommunications and Graph Theory 20
4.2 N' values in terms of RPL and d_{avg} 26

6.1 Final Constraint Size as Compared to Initial Constraint Size 61
6.2 Number of Iterations to Achieve Restorable Network Designs 62
6.3 Restorability of ICH Designed Networks by a k-shortest Paths Algorithm 66

7.1 Number of Unrestorable Links (Out of 1443 possible) when a Network Design Based on One k-Shortest Paths Restoration Scheme is Restored by Another 105

8.1 Number of Unrestorable Links (of 1443 Possible) When Network Designs are Restored by k-Shortest Paths Schemes 110
8.2 Restorability Statistics for Restoration by SHN 111
8.3 Redundancy Statistics (Σ Spare Capacity / Σ Working Capacity) 112
8.4 Growth Accommodation: Average Effects for One Random Path Addition (N=50, davg=4 Study Network, 10 Path Addition Trials) 114
8.5 Over-Restorability Statistics (All Normalized to 1443 Restorable Working Links) 115
8.6 Lines of C Programming Language Source Code 116

9.1 Number of Spans with Increased Capacity Per Provisioning Event 122

List of Figures

2.1 The 8 Successively Shortest Link-Disjoint Restoration Paths for Span Failure 1-5 ... 5
2.1 Reduction of Total Capacity by Longer Working Path Routing ... 8
2.2 Path and Span Restoration ... 9
2.4 Capacity Requirements of an SP-ring ... 10
2.5 Spare Capacity Requirements Depend Upon the Restoration Scheme ... 12
2.6 Spare Capacity Requirements From a Node-Oriented Point of View ... 13

4.1 A Non-planar Telecommunications Network ... 24
4.2 The Node Sets in the Locality of Span 1-5 for Various RPL ... 25
4.3 Constant Degree Network Topologies ... 26
4.4 A Network Topology and the Corresponding Adjacency List ... 27
4.5 A Network Topology and the Corresponding Adjacency Matrix ... 28
4.6 A Tree Structure ... 29
4.7 Heap Representation of Tree in Figure 4.5 ... 30
4.8 The Binary Heap percolate() and sift-down() Procedures ... 31
4.9 The Binary Heap delete-min() and insert() Procedures ... 31
4.10 A Depth First Search ... 33
4.11 Shortest Path From A to Each Other Node ... 34
4.12 An Implementation of Dijkstra's Shortest Path Algorithm ... 35
4.13 A Fast Implementation of Dijkstra's Shortest Path Algorithm ... 37
4.14 The metaDijkstra Procedure ... 38
4.15 Example Where k-shortest Paths Differs from Max-flow ... 39
4.16 Example Where k-shortest Paths "Flow" Depends Upon Selection Order ... 40
4.17 Cutsets Which Guarantee Restoration Flow for Span A-B ... 41
4.18 A Description of the Elements of Equation (4.4) ... 42

5.1 An Algorithm B can be Reduced to an Algorithm A by the Transformations S and T ... 44
5.2 Hamiltonian Cycles ... 45
5.3 Demonstration of Requirement of 2 Spare Links per Node ... 45
5.4 Formation of a Cycle by a Restoration Route ... 46
5.5 An Unrestorable Span ... 46
5.6 The Fundamental Cutsets Defined by the Spanning Tree (in Bold) ... 47
5.7 Demonstration of Using Localities to Restrict the Number of Cutset Constraints ... 50
5.8 Cutset Generation by Increasing RPL ... 51
5.9 Cutset Generation by Increasing Network Size with a Constant RPL ... 51
5.10 Egress Path Length Limited Cutsets ... 52

6.1 The Incident Cutsets Provide the First Set of Constraints ... 54
6.2 The Spare Capacity Placement of First Step LP ... 55
6.3 Demonstration of Adding a Cutset Constraint for Unrestorable Span 5 ... 56

6.4 The Spare Capacity Placement of Second Step LP 57
6.5 The Final Step Spare Capacity Placement 57
6.6 Incorporation of Modular Systems Into ICH Constraints 59
6.7 Time Complexity of the LP 61
6.8 Experimentally Observed Time Complexity of ICH 63
6.9 Experimental Time Complexity of ICH -- Adjusted for Number of Steps (d_{avg}=4) 63
6.10 Spare Capacity Requirements for Degree 4 Networks 64

7.1 The Restorability - Redundancy Plane 69
7.2 Finite State Machine of FS Phase of SLPA 70
7.3 The Restorability - Redundancy Plane After the FS Phase 72
7.4 An Algorithm for Adding One Spare Link 72
7.5 An Algorithm for Simultaneously Adding Two Spare Links 73
7.6 An Algorithm for Adding a Restoration Path 73
7.7 Finite State Machine for DT Phase of SLPA 74
7.8 The Restorability - Redundancy Plane after the DT Phase 75
7.9 An Algorithm for Deleting One Spare Link 75
7.10 An Algorithm for Adding One While Removing Two Spare Links 76
7.11 An Algorithm for Adding Two While Removing Three Spare Links 76
7.12 Elements of Span Localities 78
7.13 Identification of Nodes in a Locality 80
7.14 The Dijkstra-Based Restorability Calculation Procedure 81
7.15 A List of All Restoration Routes Available for Restoration of a Span After Sorting by Length 82
7.16 Sorting and Reducing Within a Common Route Length 82
7.17 Recursive Sorting and Merging Within a Common Route Length 83
7.18 The Path-Table-Based Span Restorability Calculation Procedure 86
7.19 The Path-Table-Based Network Restorability Calculation Procedure 87
7.20 Branching of Path Possibilities 87
7.21 Add One Spare Link Module of SLPA 88
7.22 Add Two Spare Links Module of SLPA 90
7.23 Add One and Remove Two Spare Links SLPA Module 93
7.24 (a) Add Two and Remove Three Spare Links SLPA Module 95
7.24 (b) Add Two and Remove Three Spare Links SLPA Module 95
7.25 Path-Table Size for SLPA 99
7.26 Quadratic Fit to Path-Table Size for Degree 4 Networks 99
7.27 Complexity of Average Total Number of Steps of SLPA Executions (d_{avg}=4) 100
7.28 Average Complexity of Extent of Search of Each Step of SLPA (d_{avg}=4) 101
7.29 Average Complexity of $R_{s,i}$ Calculation (d_{avg}=4) 101
7.30 Execution Times for SLPA Path-Table 102
7.31 Execution Time for SLPA Dijkstra 103
7.32 Execution Time for SLPA Short 103

8.1 Execution Times for Four SCP Heuristics for 36 Different Trial Designs 108
8.2 Comparison of Execution Times of ICH and Three SLPA Heuristics for Design of d_{avg}=4 Study Networks 109
8.3 Comparison of Total Spare Capacity Requirements of d_{avg}=4 Study Networks Using ICH and Two SLPA Heuristics for Design 112

9.1 Capacity Evolution With the Growth of A Network 121
9.2 Spare Capacity Requirements of a 50-node, d_{avg}=4 Network as Working Capacity Requirements Grow 123

A.1 The Telecom Canada Network [GrVe89] 136
A.2 The Indian Network [Ravi92] 137
A.3 The German Network [Vogt91] 138
A.4 The U.S. Network [Kess87] 139

B.1 20-Node Networks 141
B.2 30-Node Networks 142
B.3 40-Node Networks 143
B.4 50-Node Networks 144
B.5 60-Node Networks 145
B.6 70-Node Networks 146
B.7 80-Node Networks 147
B.8 90-Node Networks 148
B.9 100-Node Networks 149

Symbols and Abbreviations

adjL	Adjacency list.
adjM	Adjacency matrix.
C	The binary cutset matrix input to the IP (LP).
d	A constant network wide node degree.
d_{avg}	The average nodal degree of the network.
DCA	The direct cutsets algorithm
DCS	Digital crossconnect system.
DFS	Depth first search.
d_{max}	The maximum nodal degree of the network.
DT	The design tightening phase of SLPA.
EPL	Egress path length.
FS	The forward synthesis phase of SLPA.
FSM	Finite state machine.
ICH	Iterative Cutsets Heuristic.
IP	Integer program.
LP	Linear program.
N	The number of nodes in the network.
N', N'_i	The average number of nodes in a span locality, the number of nodes in span i's locality.
N_c	The number of cutsets in the IP (LP) constraints.
N_{cmax}	The specific number of cutsets in an IP (LP) constraint set.
NGA	The network generation algorithm of [DuGr91]
NP	The set of problems that a non-deterministic algorithm can solve in polynomial time.
NP-complete	A set of **NP** problems which are mutually reducible.
NP-hard	A set of **NP** problems which are at least as complex as NP-complete problems.

N_{tc}	The specific number of truncated (by localities) cutsets in an IP (LP) constraint set.
OR,OR_i	Over-restorability and over-restorability of Span i.
P	The class of decision problems that can be solved by a polynomial time algorithm.
R_n	The restorability of the network.
RPL	Restoration path length.
$R_{s,i}$, R_s	The restorability of span i, the restorability of a span.
S	The number of spans in the network.
S', S'_i	The average number of spans in a span locality, the number of spans in span i's locality.
s, s_i	The spare link capacities vector, the number of spare links on span i.
S_c	The number of spans whose restorability may have changed when assessing R_n.
SCP	Spare capacity placement.
SHN	A mesh network restoration algorithm [Grov87]
SLPA	Spare Link Placement Algorithm.
SP-ring	Shared protection ring.
SPR,SPR_i	Super-provisioning redundancy and super-provisioning redundancy of Span i.
SR,SR_i	Super-redundancy (based on restoration) and super-redundancy of Span i.
$s_{req,i}$	The spare capacity required on span i for a fully restorable network.
w'	The length N_c vector of working capacities where $w'_i=w_j$ and j is the span failure corresponding to cutset i (the i-th row of **C**).
W, W'	The number of working links in the network, the number of working links in a span locality.
w, w_i	The working link capacities vector, the number of working links on span i.
WPR	Working path routing

1 Introduction

Over the past six years, advanced methods of carrier facility restoration have evolved into a field of telecommunications research. According to [Grov89], resolving the restoration problem means to "rapidly and accurately reroute carrier signals via diversely routed spare transmission capacity in a network when a failure takes down both the working and spare links on a given span."

Emerging restoration methods based on Digital Crossconnect Systems (DCS) offer the prospect of transport networks that are restorable in one or two seconds [Grov87, YaHa88, Grov89, GrVe90, SaNi90]. Restoration plans realized by these methods closely mimic a criterion of k successively shortest link-disjoint replacement paths through the spare links on other spans of the network. The so-called mesh restorable networks that result are extremely flexible in that they inter-work with existing transmission systems, and permit integrated access to a single pool of spare capacity for both provisioning and survivability purposes.

Mesh restorable networks can be highly efficient in terms of total capacity required for restorability and can support any target level of restorability from 0% to 100%. Fully restorable mesh networks can approach a limiting redundancy of $1/(d_{avg} - 1)$, where d_{avg} is the average node degree of the network. Although mesh restoration methods have a comparatively low redundancy requirement, only optimal assignment of spare capacity to the spans of the network realizes the minimum redundancy network.

One goal of this thesis is to encapsulate the existing knowledge of Spare Capacity Placement (SCP) as a companion study to restoration mechanisms. Without some form of SCP, restoration will not be possible; and without an optimized form of SCP, the minimal capacity requirements of "mesh-type" restoration cannot be realized.

Optimal SCP can be formulated as an integer program (IP) for max-flow (but not k-shortest paths) re-routing characteristics. Although max-flow re-routing capacity will not guarantee an equivalent capacity for k-shortest paths re-routing, it has been shown that the two solutions are very similar [DuGr91]. This difference aside, the computational complexity of the direct approach renders IP infeasible for use as an SCP algorithm. In fact, it will be proven that optimal SCP is within the class of problems described as NP-hard and, as such, no polynomial time algorithm exists which can guarantee an optimal placement. It is for this reason that this work investigates approximate (heuristic) algorithms for SCP.

This thesis describes two heuristic algorithms for the task of SCP within a fixed topology and working capacity placement. The techniques investigated were both introduced in previous literature.

The Spare Link Placement Algorithm (SLPA) was developed at the Telecommunications Research Laboratories (TRLabs) in Edmonton, Alberta, in 1990 [GrBi90,GrBi91]. Because SLPA

was first introduced for operation on the specific network structures (120-node, 3.5-average node degree) defined by the Telecom Canada long-haul network, some of the techniques are not generally applicable to all network topologies. Therefore, in this work the author considers alternate implementations of SLPA. SLPA is a greedy algorithm which works in two phases, the selective addition of spare capacity until the network is restorable (forward synthesis), and the removal of excess spare capacity through redistribution (design tightening). During forward synthesis, the algorithm successively seeks to add spare capacity to the network spans where the largest increase in restorability is obtained. During the design tightening phase, SLPA accepts any redistributions of spare capacity which result in a net decrease in required spare capacity, while maintaining restorability at the desired level.

The Iterative Cutsets Heuristic (ICH) is similar to the heuristic proposed by NEC in 1990 [SaNi90]. ICH formulates SCP as a linear programming optimization problem. The min-cut max-flow theorem [FoFu56] provides constraints on restorability of a span through minimum flow cutsets of the network. The LP provides an SCP which satisfies these constraints while minimizing total spare capacity. ICH reduces the computational burden by using only a subset of the complete cutset constraints. It compensates for the missing constraints by iteratively adding constraints for only those areas of the network where the previous solution does not realize the desired restorability. This iterative process eventually converges on a feasible network design. Although ICH guarantees that a generated feasible network design is optimal, the number of steps which it requires to achieve this design is not guaranteed. Also, a certain amount of error is inherent when using a commonly available and fast linear program (LP) -- instead of the more appropriate IP -- and then forcing the returned link quantities to integral values. ICH embodies modifications and additions to the implementation of [SaNi90] in order to realize additional features, such as limiting the restoration path length (RPL) and introducing fiber capacity modularity as part of the design process.

SLPA and ICH are compared for computational complexity, total capacity required, restorability with a known restoration algorithm, and ease of implementation. In addition, a technique is introduced for updating a network design when new facilities are implemented, which respects the capacity already in place. Called joint provisioning and restorability, this is essential for any capacity management algorithm operating in a continually evolving telecommunications network.

1.1 Outline

Chapter 2 presents the mesh carrier transport network and describes the requirements of restoration.

Chapter 3 introduces the comparison measures used for this study, including time complexity, storage complexity, restorability, total network capacity, ease of implementation, accommodation of network growth, and over-restorability. The network models used in the comparison also appear here.

Chapter 4 formally presents the spare capacity placement (SCP) problem. Chapter 4 also presents a direct, exact approach to SCP, called the Direct Cutsets Algorithm (DCA). This algorithm is restrained from practical use for SCP by extremely high computation times.

Chapter 5 investigates the complexity of the SCP problem in general. After a brief introduction to the required components of complexity theory, the chapter contains a proof that the SCP problem is NP-hard. This is the primary impetus for using heuristics, rather than exact algorithms, to solve the SCP problem. The Direct Cutsets Algorithm (DCA) for SCP is then analyzed for its complexity, proving that the algorithm is NP-hard. The analysis of the direct algorithm suggests possible methods of implementing a cutset-based heuristic.

Chapter 6 presents a cutset-based heuristic, called the Iterative Cutsets Heuristic (ICH). The heuristic is described in terms of the components presented in chapters 3 and 5, then analyzed theoretically. Experimental results are presented in the following categories: time complexity, linear versus integer programming, restoration path lengths, restoration type and restorability, and system modularity.

Chapter 7 presents the other heuristic, the Spare Link Placement Algorithm (SLPA). SLPA is introduced in general terms, followed by a section which discusses implementation. The third section discloses a specific implementation and a complexity analysis. The remainder of the chapter presents experimental results for time complexity, space complexity, restoration type and restorability, and system modularity.

Chapter 8 compares the results presented in chapters 6 and 7 and also investigates accommodation of network growth, ease of implementation and over-restorability. Chapter 8 concludes with a series of recommendations for selection of an SCP heuristic.

In **Chapter 9**, SLPA is used to demonstrate joint provisioning and restorability of a network design as it evolves over time. Strategies of just-in-time capacity management and preemptive capacity management are considered. Pre-emptive capacity management is controlled by both future provisioning needs and future restorability requirements of the network.

The thesis is summarized in **Chapter 10** with suggestions for future work.

2 Mesh Transport Networks

This chapter is dedicated to describing the attributes of mesh transport networks and the properties of mesh restoration solutions, thus providing the requirements with which the SCP algorithms must comply.

2.1 The Network

Carrier transport networks are composed of nodes and spans. Nodes are digital crossconnect systems (DCS), which apply computer control to altering routes, terminating traffic, and initiating traffic. Spans are the logical connections between nodes, the pipes through which traffic can "flow." They comprise multiple links, each link carrying a fixed number of multiplexed voice or data circuits. The capacity of a link depends upon the technology used; for example, a DS-3 link transports 672 voice-circuit equivalents. The links are of two types for restoration purposes. Working links carry live traffic. Spare (redundant) links are available for future provisioning and for restoration (rerouting of traffic) of failed working links. A path is a chain of links through the network, and can be provisioned between any two nodes in the network, from the pool of spare capacity, subject to the availability of spare links.

2.2 Restoration Re-configuration

To re-establish working paths after their failure, restoration is effected by substituting a spare path (restoration path) for a portion of each failed path created by a span cut. Restoration of an entire span failure requires a separate link-disjoint restoration path for each path disrupted by the failure. A single spare link can be a component of only one restoration path.

This research assumes that only one span failure event occurs at a time. In practice, occasional violation of this assumption may occur because of the time required to physically repair a failed span. Only physical repair of the failed span returns the restoration paths to the pool of spare capacity, enabling use in another restoration event. A calculation can be performed to approximate the probability of multiple span restoration events overlapping in time. Following the work of [Grov89b] and [Krte88] on the Telecom Canada network, this calculation assumes a cable failure rate of $2*10^{-3}$ per km per annum and a manual repair time of 14 hours for a fiber span. Hence, a span that is 300 km long will be down for 8.4 hours per year (300 km * 14 hrs * $2*10^{-3}$ $km^{-1}yrs^{-1}$). This corresponds to 0.096% of the time. For a network with 200 spans, each with 8.4 hours per year of down-time, the probability of having two (or more) failures at any given time is (by realizing that this is a binomial distribution, [Proa89]):

$$P(\#\,\text{failures} \geq 2) = 1 - P(0\,\text{failure}) - P(1\,\text{failure}) = 1 - \sum_{k=0}^{1} C(200,k) \cdot (0.00096)^k \cdot (1-0.00096)^{200-k} \cong 1.6\%$$

where C(i,j) is the number of combinations of i objects chosen j at a time.

Thus, the probability of multiple spans requiring simultaneous restoration is only 1.6% even though at least one span will be failed 17% of the time. Of these events, few pairs of failures will be geographically close enough to compete significantly for restoration capacity.

2.3 Restoration as a Routing Problem

The routing required for restoration has attributes that differ from other common routing mechanisms such as packet routing or call routing. The discussion here will summarize some of the related material presented in [Grov89].

Restoration paths are link-disjoint replacement paths substituted for segments of failed paths. This mechanism assigns a distinct restoration path for each failed working path segment. For example, Figure 2.1 presents the k successively shortest link-disjoint restoration paths for a span failure between nodes 1 and 5 through an example network. A transmission network is described as a multi-graph, because multiple links exist on a span and each link is treated as a separate entity. The aspects of link-disjointness in a multi-graph are not characteristic of other routing problems. Most other routing processes operate on simple graphs where only one link is available on each span and routings are not mutually link-disjoint. Links may take the form of a grouping of trunks, but the access is to the group instead of to the individual entity.

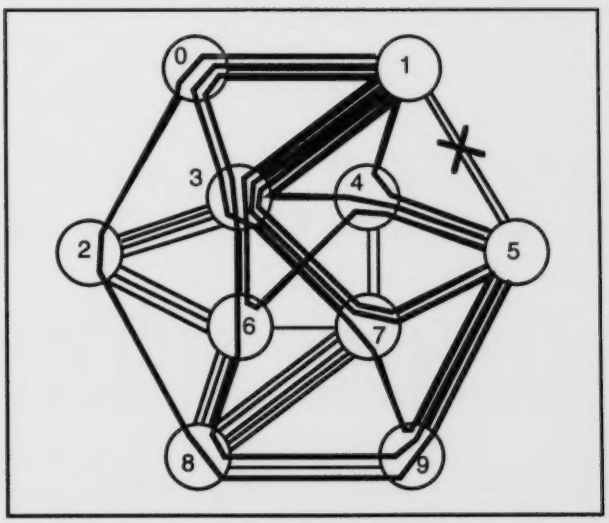


Figure 2.1 The 8 Successively Shortest Link-Disjoint Restoration Paths for Span Failure 1-5

The packet routing mechanism establishes a route for each source-sink pair by transmitting packets through intermediate nodes. Each source-sink pair stores the sequence of nodes through which this route was established in a "routing table." The routing table dictates the route over which all subsequent packets for that source-sink pair will transit. Packet routing associates only one "pipe" with each span and, therefore, a packet network is represented as a

"simple graph." In packet routing, any number of source-sink pairs can share a span within their routing table entries.

Similarly, a dynamic adaptive call routing mechanism routes an offered call attempt through a sequence of trunk groups, from the source node to the sink node. This mechanism is similar to packet routing in that it operates in a network represented by a simple graph. Each trunk group is accessed as a single entity and, therefore, after a route is established, the mechanism arbitrarily assigns a trunk from each trunk group to carry the requested call.

Call routing and packet routing have other attributes that are different from the restoration routing mechanism; however, for this thesis, the simple graph network representations leading to non-link-disjoint routing is the important distinction from the restoration routing problem. The restoration problem requires identification of multiple link-disjoint paths. As a consequence, a single route substitution restores networks that implement simple graph routing; but, in the carrier transport network, only a distinct link-disjoint path substitution for each failed link provides restoration.

2.4 Topology Design and Working Path Routing

Optimal network design for restorability requires three design phases: network topology, working path routing (WPR), and spare capacity placement (SCP). Although optimal network design can only truly result from simultaneous consideration of all three of these phases, computability dictates the separation of these problems.

A network topology must have graph-connectivity greater than one in order for restoration to be possible, otherwise a span exists whose removal disconnects the network and therefore leaves no restoration route. Appendix A presents actual network topologies which are currently deployed. These networks have average node degrees of three or more and provide enough route diversity to support restoration by rerouting (except some of the outlying areas of the Indian network). [MaWi91] addresses specific aspects of restorable network topology design such as efficient placement of spans and nodes. Also, the use of matroids to ensure restoration capabilities which are equivalent for max-flow and k-shortest paths path selection is proposed as a future investigation. Otherwise, this thesis does not consider network topology design further.

The second step in designing a carrier transport network is the routing of working paths. The network requires a minimum total working capacity when all working paths are placed on the shortest possible routes -- this is shortest path routing. However, in designing a network that must be restorable, shortest path routing does not always result in the minimum total (working + spare) capacity. Thus, an optimal capacity design procedure requires the simultaneous allocation of both working and spare capacity, which ensures that the working paths are routed over

relatively short routes as well as being easily restorable, and results in maximal sharing of the network's redundant capacity.

The problem of optimum SCP within a network with fixed working capacity and topology is difficult in itself. Simultaneously attempting to optimally route the working capacity exacerbates the problem. In fact, a feasible heuristic to solve WPR and SCP, optimally and simultaneously, has not been proposed to date. To reduce the complexity of this dual-faceted problem, [GrBi90] took a sequential approach which first routes the working paths, with knowledge of the placements that cater to good later sharing of spare capacity for restorability. Spare capacity is then optimally assigned based on these working path routings. In studies performed on the Telecom Canada network, this method resulted in a decrease in total capacity of approximately 3% from the network designs with shortest path routing of working capacity. Such small improvements do not justify the complexity added to the design procedure. This thesis does not investigate the issue of optimal WPR beyond the summary provided in the remainder of this section.

The algorithm used in [GrBi90] to route working paths exploits the end-node limited condition of mesh restorable networks (see Section 2.6). From a node-oriented perspective, the minimum redundancy is achieved when the working capacity is uniformly distributed among the spans incident to a node. A nodal balance algorithm evaluates potential routings of a working path for their contribution to nodal balance in the network. It calculates this contribution as the sum of the nodal imbalance metrics associated with each node along the path. At a single node, the metric is the sum of working capacities on the spans being considered for the new path minus twice the average capacity of all spans. Thus, a negative metric value is desirable because it indicates an overall increase in nodal balance of working capacity. At each iteration, the working path routing algorithm provisions the path with the smallest sum of nodal balance metrics. The algorithm routes the shortest paths first, so more knowledge of the network will be available when routing the longer, more flexible paths.

[GrBi90] observed that minimum total capacity occurs with route lengths of 1.0 to 1.154 times longer than shortest paths routing. [GrBi90] repeated the whole network design (WPR and SCP) for factors ranging from 1.0 to 1.3 to identify its optimal value for these networks.

The remainder of this section provides an example of how a longer working path route can result in smaller total capacity. In the sub-network of Figure 2.1, some spare capacity has already been provisioned for restoring spans not depicted in the figure. For simplicity of analysis, there is no working capacity yet routed on any of the spans included in the sub-network; therefore, the nodal balancing requirement cannot be readily observed in this network. The network requires an additional working path between Node A and Node C. Consider the routing alternatives A-B-C, A-E-C, and A-E-F-C. When provisioning these new working links, the network

requires additional spare capacity to maintain restorability. The addition of a working link to spans A-B, B-C, or E-C requires extra spare capacity for restoration; however, accepting the longer route (A-E-F-C) avoids these trouble spans and no requirement for additional spare capacity exists. Hence, in this case, choosing the longer route minimizes the total number of links added.

2.5 Path or Span Restoration

Which segment of a failed working path should be replaced with a restoration path segment? The replaced segment must include the failed span; however, it can also include any number of the other links of the original path, on either side of the failed span.

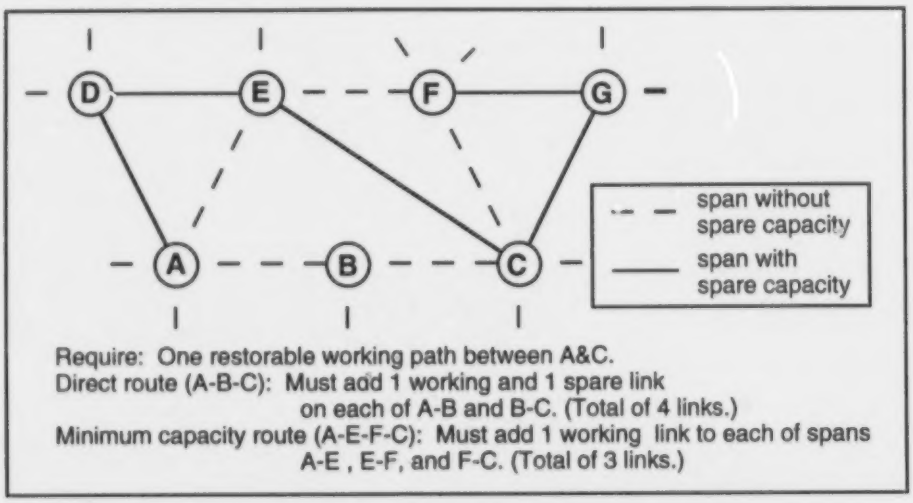
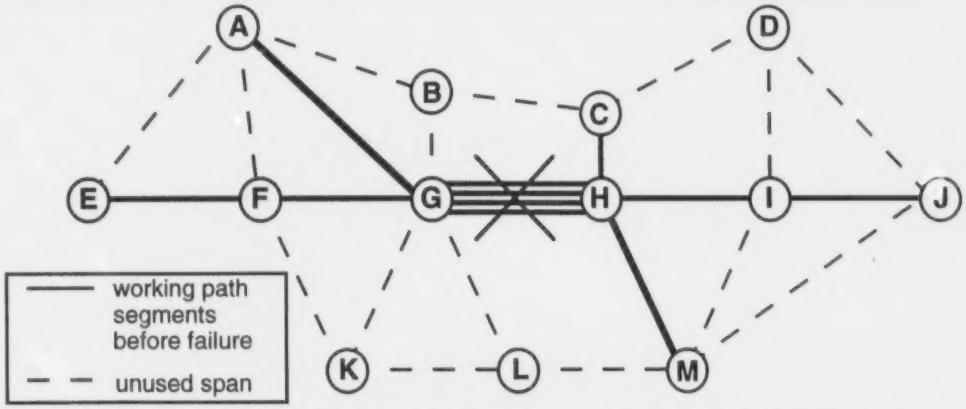

D
E
F
G
A
B
C
span without spare capacity
span with spare capacity
Require: One restorable working path between A&C.
Direct route (A-B-C): Must add 1 working and 1 spare link on each of A-B and B-C. (Total of 4 links.)
Minimum capacity route (A-E-F-C): Must add 1 working link to each of spans A-E , E-F, and F-C. (Total of 3 links.)

Figure 2.2 Reduction of Total Capacity by Longer Working Path Routing

Fast restoration algorithms developed to date have provided restoration paths between the two nodes adjacent to the failed span. This is called span-restoration. It is the only restoration that guarantees to replace the same logical failed segment for every failed path, and therefore provides a reduction in restoration complexity by allowing a single source-sink pair search for all restoration paths. For example, Figure 2.2 depicts a network in which Span G-H fails. Here, solid lines indicate the failed working paths. Span-restoration is the only alternative whereby paths between a single source-sink pair of nodes can restore all failed working paths because paths A-G-H-C and E-F-G-H-I-J only share nodes G and H.

Path-restoration seeks to replace failed paths with entirely new paths from source to sink. Path-restoration generally requires a different source-sink restoration path for each failed working path. Therefore, it requires multiple path searches to establish the restoration paths, and results in a decrease in restoration speed. However, path-restoration can offer the most efficient use of

network capacity, because it is possible to avoid the area of the network where the failure occurred and capacity is at a premium. In Figure 2.2, when path A-G-H-C fails, path-restoration may route the path over A-B-C and free the links on the original route for use as spare capacity for another failure. When restoring the same path with span-restoration, the shortest restoration route is via nodes B and C, making the new path A-G-B-C-H-C. Thus, near the failed span where spare capacity is most valuable, path-restoration has a lower capacity requirement than span-restoration.



Path Restoration: Reroute original path via an alternate route between end-nodes of path (A-G-H-C path before failure can be rerouted A-B-C after the failure, releasing the spare links on spans A-G and C-H).

Span Restoration: Reroute failed link of original path between end-nodes of failure (A-G-H-C path before failure is rerouted to A-G-B-C-H-C path after failure).

Figure 2.3 Path and Span Restoration

Of course, any intermediate node pair, between the end-nodes of the failed span and the end-nodes of the path, can alternately be selected for anchoring restoration paths. In specific situations, an intermediate selection could provide a balance between the benefits of span and path restoration.

The remainder of this thesis assumes span-restoration for networks designed with the SCP heuristics. There are two reasons for this decision. Published restoration techniques that claim restoration speeds under 2 seconds[1] use span-restoration. The extra complexity of path restoration makes it less likely that a path restoration algorithm, if developed, would meet the same real time objectives. And networks designed for span-restoration will contain enough

1 The call dropping threshold is commonly 2 seconds. Therefore, restoration which can be completed within two seconds will avoid a loss of service.

capacity to perform path restoration, but the opposite statement is not necessarily true. Therefore, assuming span-restoration will upper-bound redundancy requirements.

2.6 Mesh Versus Ring Restoration

The two SCP algorithms investigated here seek to design mesh-type restorable networks. However, some researchers have investigated ring-type restorable networks [LaAb86, TsCo90, Wrob90, McGo88, CaMo89, WuHa89, FlOx89, Flan90]. The scope of the current research does not allow for a full discussion of the benefits and liabilities of each restoration method, but this section will present a theoretical comparison of redundant capacity requirements. This section will introduce ring-type restoration, briefly describe the network design issues associated with rings, then discuss their redundancy requirements. A theoretical discussion of redundant capacity requirements for mesh restoration will follow. This section concludes by comparing the redundant capacity requirements associated with ring and mesh restoration methods, thereby further motivating the study of optimum mesh-restorable networks.

Ring-type restoration is characterized by restoration over a single predefined route. The spans of the predefined restoration route, coupled with the failed span, comprise a restoration ring. The most efficient type of restoration ring, the shared protection ring (SP-ring), facilitates shared access to the spare capacity of the ring for the restoration of any span in the ring. Figure 1.1 provides an example of a single SP-ring. Because all spans on the ring must be restorable via the remainder of the ring, the capacity of the largest working span dictates the spare capacity required on the other spans. The spare capacity required on any span is the maximum of the working capacities on any other span in the ring.

In the network of Figure 2.4, the spare capacity required on span A-B will be the greater of the working capacities on spans A-C and B-C. The spare capacity assigned to A-B will be available for restoration of both of the other two spans on the ring, but will not be available to any other spans in the network which are not included in this ring.

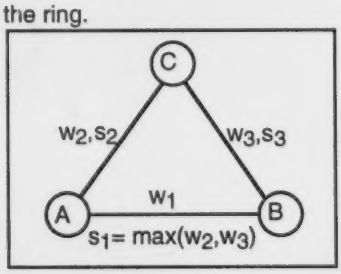


Figure 2.4 Capacity Requirements of anSP-ring

The optimal design of ring restorable networks must incorporate three aspects of network design: grouping spans into restoration rings, routing the working paths, and placing the spare capacity for restoration. The redundant capacity required for restoration is minimized with rings which have balanced working capacity on all spans. Thus, routing working capacity and mapping nodes and spans into rings must simultaneously seek to balance the working capacity of the rings. Balanced working capacity allows the most efficient allocation of spare capacity among the

spans of the ring. In a ring with identical working capacities on each span, the full spare capacity of the ring is utilized in every restoration event. When the optimum ring definitions and working capacity routes have been identified, the spare capacity requirements will be defined as a consequence. Thus, the problem of optimum network design must include more than just the step of calculating spare capacity requirements; it must start with defining the network sub-rings (number, size and location) and routing the working paths through the rings from source to destination.

The most efficient ring requires as many spare links on each span as there are working links. Therefore, the required spare capacity is 100% of the working capacity of the network: Redundancy is 100%. Although this theoretical minimum redundancy is possible, other restrictions on the design usually prevent it from obtaining this optimum redundancy. For instance, in most ring designs, it is required that rings join to neighboring rings at more than one node, which often results in a shared span. Each ring requires spare capacity on the common span, resulting in reduced efficiency.

Ring designs restrict restoration to a single predefined route for each span failure. A more efficient restoration method would impose no restriction on use of the network's spare capacity; it would employ a network-wide sharing of spare capacity. Mesh restoration achieves this.

Mesh-restorable network design requires that the network's spare capacity satisfies two conditions to provide restoration for each span. There must be enough spare capacity on spans adjacent to the failed span to support the required number of restoration paths between the two nodes on both ends of the failure. There must also be spare capacity through the remainder of the network for the immediately-adjacent links to be joined into complete restoration paths. If the network design is constrained by the first condition, restoration is said to be "end-node limited." Otherwise, restoration is said to be "bodily limited." It has been found in [GrBi90,GrBi91] that real transport networks are primarily end-node limited. As a result, we can formulate a simplified node-oriented approach to spare capacity requirements.

Consider one node with degree of d. All working capacity adjacent to this node is restorable if the sum of the spare capacities on the other d-1 spans is at least as large as the working capacity on any failed span. In a network with balanced working capacity and an end-node limited condition, full restoration can be accomplished with a redundancy of 1/(d-1). For example, at a node with only 2 spans, a redundancy of 100% is required to provide restoration. However, at a node with 5 spans with equal working capacities on each, only 25 % redundancy is required to provide full restoration. Thus, mesh restoration exploits the connectivity of the network to reduce the required restoration capacity. Ring capacity requirements are the two-span extreme of the mesh capacity requirements.

The amount of redundant capacity required to achieve restoration represents a major difference between mesh and ring design as illustrated by Figure 2.5. Using ring restoration, the required spare capacity is established on a span-oriented basis, whereby each span must have enough spare capacity to restore any other span in the ring. In mesh restoration, the spare capacity can be set on a node-oriented basis. Surviving spans can team up to provide restoration capability for the working capacity of a failed span.

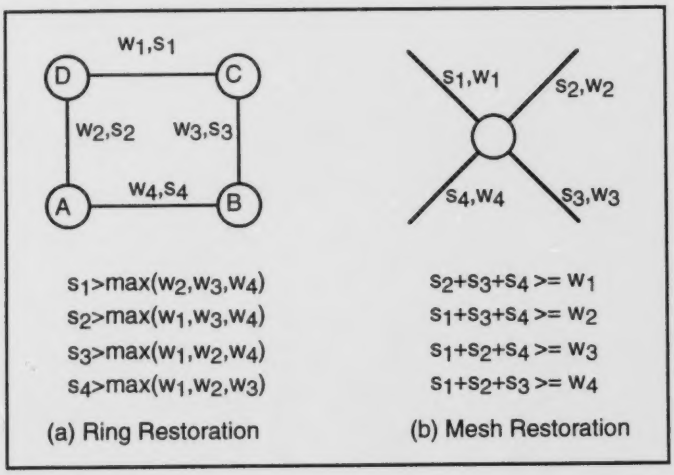


Figure 2.5 Spare Capacity Requirements Depend Upon Restoration Scheme

Figure 2.6 takes a node-oriented perspective to the spare capacity requirements for both ring and mesh restoration designs having node degrees of 4. The ring-based network was assumed to have an optimal design, with working capacity balanced between all spans of a ring. Two rings touch at the node depicted. Each of the rings is balanced with each span having a working capacity of 6 links, facilitating restoration with the other 6 links. The total redundancy required meets the optimal limit of 100%. The working capacity of the mesh network is also balanced between spans in order to optimize the design. Because all spans share equally in the restoration effort, the spare capacity required on each span is 1/3 of the working capacity. The mesh-restorable design uses the same 4 lightwave systems as the ring-restorable design, but 67% of the capacity carries working traffic and only 33% of the capacity is redundant. [GrBi90] provides more examples of capacity comparisons between ring and mesh restoration for particular structures present in the Telecom Canada network, which has been likened to a ladder topology.

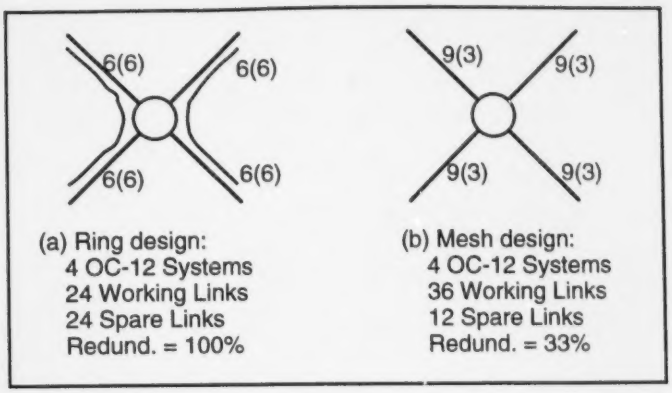


Figure 2.6 Spare Capacity Requirements From a Node-Oriented Point of View

Mesh restoration requires less redundant capacity than ring restoration. Because this thesis seeks to identify algorithms which minimize spare capacity, mesh restoration is desirable.

3 Performance Criteria and Experiments

This chapter introduces the specific characteristics that are desirable in an SCP heuristic. In subsequent chapters, the author evaluates ICH and SLPA on these characteristics. Section 3.7 presents the networks used to evaluate these characteristics through experiments.

3.1 Computational Complexity (Time and Space)

For each algorithm, time complexity will be characterized both theoretically and experimentally. The theoretical analysis will investigate worst-case and average-case execution times in terms of the following network size parameters: (a) number of nodes (N), (b) number of spans (S), (c) average node degree (d_{avg}), and (d) restoration path limit (RPL).

The experimental execution time results are obtained from compiled 'C' language source code on the SUN SPARC 2™ RISC-based computing platform with execution speed of 28.5 MIPS. Diskless workstations were used with 16 Mb main memory and 48 Mb swap space. All experiments use RPL=10, where required. The heuristic algorithms investigated here can be compared experimentally because a single programmer implemented all of them. The major contributors to the complexity of the heuristics are Dijkstra's shortest path algorithm in SLPA, which has been very carefully coded, and a Linear Program (LP) in ICH, which is solved with commercially available software.

In the original implementation of the SLPA heuristic, [GrBi90] reduced execution time by precompilation of the restoration routes into a path-table. For this implementation, both time and space complexity analysis are required.

3.2 Restorability

Restorability of each span is the most important constraint imposed upon SCP algorithms. A span-cut i between nodes (s,t) results in w_i working paths being interrupted. Restoration of this span is achieved by rerouting the interrupted links over k_i link-disjoint replacement path segments through the spare links on surviving spans. The span-cut is fully restorable if the number of possible restoration paths, k_i, is greater than w_i. Span restorability is defined as:

$$R_{s,i} = \frac{\min(w_i, k_i)}{w_i} \tag{3.1}$$

The restorability of a network as a whole is the ratio of the number of working links in the network that are restorable to the total working capacity (in links) of the network.

$$R_n = \frac{\sum_{i=1}^{S}[\min(w_i, k_i)]}{\sum_{i=1}^{S}[w_i]} = \frac{\sum_{i=1}^{S}[R_{s,i} \cdot w_i]}{\sum_{i=1}^{S}[w_i]} \tag{3.2}$$

where S is the number of spans in the network.

Here, the network restorability target is 100%. This objective maximally stresses the SCP algorithms. Moreover, the ability to achieve network designs that are 100% restorable is an important feature of any heuristic which claims to design restorable networks. The heuristics will also be evaluated for ability to accommodate restorability objectives that are less than 100%.

3.3 Total Network Capacity

A common format for expressing amount of spare capacity is redundancy of the network, defined as follows:

$$Red_n = \frac{\sum_{i=1}^{S}[s_i]}{\sum_{i=1}^{S}[w_i]} \tag{3.3}$$

where w_i and s_i are respectively the working and spare capacities of span i.

This research does not use this definition in its most general form because of the significant discrepancies that different working capacity routings can introduce. As was discussed in Section 2.4, working capacity can be routed in a manner that increases the sharing of spare capacity and, therefore, decreases the requirement for spare capacity. Taking this method to the extreme, it can increase working path lengths while decreasing the required number of spare links to achieve restorability. Such designs will have a lower redundancy by formula (3.3), even though the total capacity and the spare capacity have been increased. Thus, a measure of redundancy more suitable to SCP comparison is normalized to the amount of working capacity implied by a shortest path routing.

$$Red_n = \frac{\sum_{i=1}^{S}[s_i]}{\sum_{i=1}^{S}[w_{i,\text{shortest path}}]} \tag{3.4}$$

This normalized redundancy figure is required for comparing networks of varying sizes. However, when comparisons are based on a common network, the issue of what value to use as the denominator can be avoided entirely by adopting total network capacity as the figure of merit. Therefore, this thesis uses total network capacity wherever possible for evaluating the optimality of designs; or, total spare capacity when common working capacity placement between designs exists.

3.4 Ease of Implementation

This thesis also compares heuristics for ease of implementation, a preference given to algorithms with concise specifications, measured by the amount of computer source code

generated. With an easily specified heuristic, a third party can quickly and accurately produce his/her own version.

3.5 Accommodation of Network Growth

An SCP heuristic should consistently produce a near-optimum network design, both in terms of minimizing capacity and meeting restorability objectives. An additional requirement of network evolution arises after initial design implementation, when the network capacity grows beyond the original design: In the face of service growth, how can an SCP heuristic evolve a previous network design to accommodate the new growth, while optimizing the design in the process? This is an essential aspect of real world networks that grow continually and should evolve gracefully rather than being periodically redesigned from the ground-up. Specifically, during the evolution to a new restorable network state it is preferable if a minimum of spans are affected. This research considers two approaches for updating an SCP solution as the underlying "working" capacity of the network grows. These methods are called "incremental" and "ground-up."

In **incremental** growth accommodation, span capacities can only increase. Existing capacity is never removed or relocated. This avoids the expense associated with removing or relocating existing capacity. This form of growth accommodation is easily implemented into any SCP heuristic by solving a smaller SCP problem in a network of the same span topology but in which $w_i > 0$ only where non-restorable spans exist in the full network after growth. The spare capacity already present in the network is unchanged during the evolution. The new spare capacity requirements are obtained by adding the spare links to realize full restorability in the subnetwork design to the previous spare capacities.

The **ground-up** growth accommodation strives to optimize the new network design rather than just the new layer of capacity. Therefore, this will result in a more efficient design in terms of capacity, but will often remove or relocate capacity. For some heuristics there may not be a method of performing ground-up growth accommodation while still avoiding widespread redistribution of capacity. In order for a heuristic to achieve an acceptable form of ground-up growth accommodation, the location of capacity in the previous network design must carry some weight during the evolution process.

3.6 Over-Restorability

A companion measure to restorability is the over-restorability of the network. Over-restorability is a measure of the ability of the network design to maintain restorability while provisioning additional working paths from the pool of spare links in the network.

In the SCP problem, spans that have higher working capacities drive the network design, because they require larger restoration capabilities. Some spans may lie in the "shadows" of

these design-driving spans and enjoy access to their large pool of spare capacity yet have a smaller working capacity. This means that additional working capacity may be added to the smaller spans without any requirement for additional restoration capacity. The total number of restoration paths that exist in the network (reflecting the maximum number of working links that could be restored if so provisioned) as a fraction of the working links present represents over-restorability, or capability for increased working capacity without restorability loss.

$$OR_n = \frac{\sum_{i=1}^{S} [k_i]}{\sum_{i=1}^{S} [w_{i,\ \text{shortest path}}]} \tag{3.5}$$

where $w_{i,\text{shortest path}}$ is the number of working links on span j when all working paths are shortest path routed, and k_j is the number of restoration paths available for restoring a failure of span j. When comparing networks with identical working capacities and topologies, the maximum number of restoration paths available is an equivalent measure of over-restorability.

3.7 Experimental Network Models

Experimental network topologies were generated to exercise the heuristics over a range of networks varying in number of nodes and average node degree. Testing the heuristics on generated random networks allows more freedom to vary network size and degree systematically than could be provided by known network topologies. However, the known network topologies of Appendix A did provide important network characteristics used in generating the random networks. The algorithm used to generate the networks is called the network generation algorithm (NGA). The following sections summarize the description of NGA from [DuGr91] for those functions used here.

3.7.1 Network Topology Characteristics

The network generation algorithm can vary the characteristics of the average node degree, the number of nodes in the network, and the maximum magnitude of spatial separation between adjacent nodes. NGA also includes a technique for tailoring the distribution of the individual node degrees to more closely approximate the characteristics of a known network topology, but this technique was not used here.

3.7.2 Generation of Test Networks

The user of NGA specifies the target number of nodes and spans in the topology to be generated. The first step performed by NGA is randomly inserting nodes on a (GxG) grid, where G is calculated based on the target number of nodes in the network, as follows: $G = (1.8 * N)^{0.5}$.

After NGA establishes the node positions, it adds spans to the network by the following process: Randomly select a node. Define a point P that is 1.55 grid spaces away from the

selected node at a random angle. ([DuGr91] selected the value of 1.55 to ensure one grid space separation (either diagonal or width) for most adjacent nodes.) Add a span between the selected node and the closest node to point P. This procedure ensures that adjacent nodes are within 2 grid separations of each other. Repeat this process of adding spans until reaching the target average node degree. The design process ends when NGA reaches the objective, or if it determines that no more progress is possible by attempting to add more spans. After completing the span addition phase, NGA removes any nodes that have a degree of one from the network design.

3.7.3 The Networks

Table 3.1 contains a summary of network parameters for the study networks used throughout this thesis. The table contains the network sizes in terms of N and S ($d_{avg} = 2 \cdot S/N$). Appendix B shows the corresponding network topologies.

Table 3.1 Study Network Parameters

Name	N	S	Name	N	S
n20s30	20	30	n60s150	60	150
n20s40	20	40	n60s180	60	180
n20s50	20	50	n70s105	68	103
n20s60	20	60	n70s140	70	140
n30s45	30	45	n70s175	70	175
n30s60	30	60	n70s210	70	210
n30s75	30	75	n80s120	81	122
n30s90	30	90	n80s160	80	160
n40s60	39	59	n80s200	80	200
n40s80	40	80	n80s240	80	240
n40s100	40	100	n90s135	90	137
n40s120	40	120	n90s180	90	180
n50s75	46	71	n90s225	90	225
n50s100	49	99	n90s270	90	270
n50s125	50	125	n100s150	93	143
n50s150	50	150	n100s200	100	200
n60s90	58	88	n100s250	100	250
n60s120	60	120	n100s300	100	300

4 SCP Problem Formulation

The problem of optimum spare capacity placement (SCP) for a mesh transport network is best described in terms of algorithmic graph theory concepts. This chapter introduces the fundamental graph theory concepts and algorithms that are required in a formal specification of the SCP problem. In Chapter 5, analysis of the complexity of the SCP problem uses these graph theoretical concepts.

4.1 Introduction to Graph Theory

In this thesis the author refers to the components of the network in the terminology of the telecommunications community. The primary differences in terminology between telecommunications and traditional graph theory appear in Table 4.1. The next few paragraphs describe a telecommunications network in graph theoretical terms, to facilitate comparisons to graph theory literature.

Table 4.1 Terminology Comparison of Telecommunications and Graph Theory

Telecommunications	Graph Theory
Network	Multi-graph
Node	Vertex
Link	Edge
Span	Set of all edges with common vertices
Path	Chain
Ring	Cycle

A graph is a collection of vertices (nodes) and edges (links) which connect the vertices. The edges may be assigned a direction of flow or they may be undirected. A multi-graph is a graph with more than one edge between any pair of vertices. For the purposes of this research, a telecommunications network is an undirected multi-graph in which each edge provides flow in both directions; this symmetric digraph (transmission direction is important) has an edge (i,j) for every edge (j,i). Hence, a telecommunications network is also balanced, meaning that each node has an equal number of outgoing directed edges as incoming directed edges.

Links are incident to a node when they terminate at it. Adjacent links are incident to a common node. Similarly, adjacent nodes share a common incident link (span) and they are called the end-nodes of the link (span).

The connectivity of a network is the minimum number of spans whose removal results in a network with multiple components (internally connected subgraphs). In this thesis, connectivity is not the average number of spans incident on a node, as in some literature. Here, the number of spans incident on a node is its degree, although in graph theory the degree of a node usually refers to the number of edges incident on a node.

In the graph theoretical sense, network connectivity is an important part of network design for restorability. In order for a network to be 100% restorable, its connectivity must be at least 2. In a network with a connectivity of 1, there is a span whose removal disconnects the network and makes the network unrestorable.

4.1.1 Cuts and Cutsets

A cut of a network is a partitioning of the set of nodes into two subsets. A cutset is the group of spans that are incident to a node in each of the node sets of a cut. Also, the removal of the spans of a cutset from the network must produce exactly two components.

4.1.2 Efficiency Analysis

The efficiency of an algorithm can be determined empirically or theoretically. The empirical (or *a posteriori*) analysis involves implementing the algorithms and comparing results on different problem instances. The theoretical (or *a priori*) approach consists of mathematically determining the quantity of resources (execution time, memory space, etc.) needed by each algorithm, as a function of the size of the problem instances considered ([BrBr88]). Both of these efficiency measures will be used in this thesis.

A common way to express the "complexity" of an algorithm (the quantity of resources required) is asymptotic notation. The analysis performed herein primarily uses "the order of" category of asymptotic notation. This provides an upper bound, to within a multiplicative constant, of the complexity of the algorithm. For a more specific statement of this notation, consider an algorithm, B, which seeks to solve a problem instance of size n. Here, f(n) is a specific mathematical function of n that is being used to bound the complexity of the algorithm (eg. $f(n)=n^2$).

> Algorithm B is "the order of f(n)," or O(f(n)), if a positive real multiple of f(n) exceeds the amount of resources consumed by B, provided that n is sufficiently large (greater than some threshold n_0). [BrBr88]

For example, if B is $O(n^2)$, then the number of operations (or storage requirements, etc.) performed by B cannot exceed $c \cdot n^2$ in any problem instance where $n>n_0$, for some positive real constant c. Thus, it also follows that if B is $O(n^2)$, B is also $O(n^3)$, but the latter statement is a looser bound describing B.

This thesis does not use the other forms of asymptotic notation, such as specifying lower bounds or the combination of lower and upper bounds.

4.1.2.1 Why Asymptotic Analysis?

Asymptotic analysis excludes the constant multiplier (c), not to imply that the constant is unimportant but, rather, reflecting the knowledge that the constant only affects the scale of the

calculation time and does not affect the rate at which the calculation time increases with the size of the problem. Faster computers and better software engineering (among other things) can compensate for the constant multiplier, but primarily the asymptotic complexity determines the magnitude of the problem solvable by the particular technique. Of course, exceptions to this rule exist where the constant multiplier is large enough to restrict feasible use of a low complexity algorithm (eg. polynomial LP implementations [Schr86]). Experimental measurements can provide an estimate of the constant multiplier, thus also providing estimates of actual execution time for all problem instances.

In this work, algorithms for SCP are carefully designed to minimize their asymptotic (time) complexity, because telecommunications network sizes can be very large. Only after designing the algorithm for low asymptotic complexity is an attempt made to minimize the constant multiplier.

The primacy of asymptotic complexity over implementation details (which alter only the constant multiplier) is illustrated by the following amusing anecdote.

> In an Indian folk tale called "The King's Chessboard," the King forces a wise man to accept a reward for services rendered. The wise man exposes the King's lack of background in mathematics by requesting an exponentially increasing amount of rice as payment: one grain of rice on day one (the first square of his chessboard); two grains of rice on day two (the second square of his chessboard); four grains of rice on day three (the third square of his chessboard); and so on. Payment is complete after 64 days when 64 payments account for each square of the King's chessboard. Each day, the amount of rice doubles and, therefore, the request is $O(2^n)$ or exponential in amount of rice. The King recognizes the developing pattern only on day 31 when 16 wagon-loads of rice are paid. He summons the royal mathematicians, who calculate the total amount of rice promised to be 549 755 830 887 tons (which we know is 2^{64}-1 grains). [Birc42]

In the tale, the constant multiplier is a single grain of rice -- indeed, the smallest imaginable fee from the King's perspective. Now, the primacy of the asymptotic function can be demonstrated by assuming that the King had promised a single molecule from a grain of rice on the first day. If the constituent molecules of rice are larger than 2^{-64} times the size of a grain of rice (which is the approximate size of a carbon atom), the same amount of rice would be owing before the number of days reaches 128 (a mere doubling of problem size).

In computing problems, a single machine instruction on the computing platform available is the smallest conceivable multiplier -- the grain of rice. Therefore, using an algorithm with an exponential asymptotic execution time of $O(2^n)$, given even a single machine instruction that can solve the entire problem executing at a speed of 1 ns, the total execution time for a 64-node network is 2^{64}·1 ns or 585 years. Again, reducing the constant multiplier by a factor of 2^{-64} only

allows solution of a 128-node network in the same 585 years. Clearly, an algorithm that is $O(2^n)$ cannot solve this problem for an arbitrary network size, independently of the constant multiplier.

As is often true for important real-world problems, the direct approach to solving the SCP problem will be shown in Chapter 5 to be exponential in nature and, thus, proposed exact solutions make the same promise as the King. Armed with the King's knowledge, the author seeks a heuristic algorithm that finds an approximate solution to the SCP problem in a time which increases polynomially with network size.

This extreme example of exponential asymptotic complexity serves to point out the usefulness of asymptotic analysis. However, it is generally accepted that only algorithms with low degree polynomial asymptotic execution times are feasible for use.

4.1.2.2 Parameters of Complexity

For the SCP problem, execution time is the primary complexity metric. Thus, the analyses of algorithm "complexity" (the amount of resources used by an algorithm) always consider the number of operations required by the algorithm or the algorithm's execution time.

Often, the execution time of repetitive tasks within an algorithm can be decreased by preconditioning the task. In preconditioning, an initial computation is performed which reduces a repetitive task's execution time. Generally, the preconditioning information requires extra storage space and, therefore, preconditioning effectively trades space requirements for execution speed. In some algorithms that are implemented here, storage space requirements introduced by the preconditioning impact the feasibility of the method. In these situations, complexity analyses consider both space and time.

4.1.2.3 Average-case Analysis

In some cases, analysis of algorithm complexity using worst-case asymptotic complexity is not indicative of expected complexity. For this reason, this thesis complements worst-case complexity with experimental observations of execution times for a wide range of input sizes. These experimental results reveal the "average-case" behavior of the algorithm under test. Here, $E(f(n))$ denotes the average-case complexity (derived from experimental observations) where $f(n)$ indicates the expected complexity (to within a multiplicative constant) as a function of the network size.

4.1.3 Quasi-Planarity

A planar network is one that can be arranged on a plane surface such that no spans cross and no two nodes have the same location [Gibb85]. This is akin to the concept of a planar circuit in electrical circuit theory. Generally, a telecommunications transport network is nearly planar, but exceptions similar to that depicted in Figure 4.1 are reasonably common. Quasi-planarity is a defining characteristic of the networks considered here.

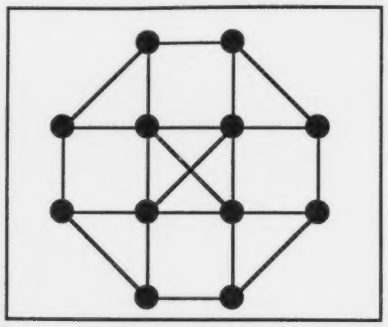


Figure 4.1 A Non-planar Telecommunications Network

The concept of quasi-planarity is an important part of the analyses that follow. One important task of this thesis is to define an SCP algorithm that can quickly (at least daily) recompute the spare capacity requirements. Because there is no limit to the size of telecommunications networks, complexity analysis is used to predict an algorithm's usefulness over a broad range of network sizes. However, a common form of analysis of algorithms that operate on graphs is to assume the worst case complexity, in which every node is or can be logically adjacent to every other node. This form of analysis is not applicable to quasi-planar networks, because nodes are generally only adjacent to those nearby in grid space. Thus, the complexity analyses here make assumptions based on the quasi-planarity occasionally approximated as planarity of the networks implemented.

4.1.4 Locality Information

A span's locality includes all spans and nodes that can topologically contribute to restoration of that span under some routing criterion for restoration. The size of span localities is dependent upon the allowed restoration path limit (RPL) and the network's topology. N'_i and S'_i denote the number of nodes and spans, respectively, in the locality of span i. Figure 4.2 depicts the sets of nodes which are included in the locality for a particular span for various RPL values.

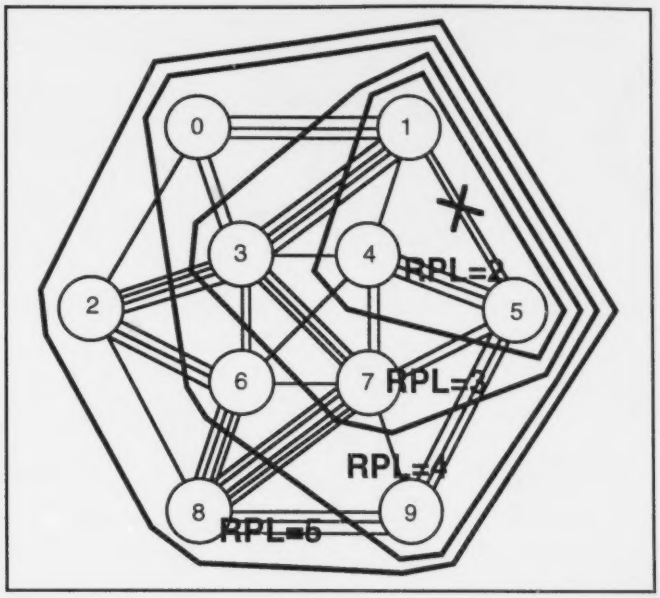


Figure 4.2 The Node Sets in the Locality of Span 1-5 for Various RPL

Algorithms that operate on networks generally have complexity which depends upon N, the number of nodes in the network. However, algorithms that have objectives based on restorability (R_S), such as the SCP algorithms, may only need to operate on span localities to determine R_S. Therefore, as the network outside the locality of span i grows in size, the complexity for calculating $R_{S,i}$ need not change; it still depends only on N' and S'. When N is much larger than N'_i, N'_i is a constant as the network grows in size. Therefore, localities provide a valuable constraint for reducing asymptotic complexity of SCP heuristics.

The concept and usefulness of localities is strengthened by near-planarity of the study network and by restoration path length (RPL) limitations. Quasi-planarity restricts the node degrees, d, observed in these networks, because nodes are only adjacent to relatively few neighboring nodes and are not adjacent to nodes on the other side of the network. The replacement paths used in network restoration are generally restricted in length by network requirements, such as echo control. With RPL and d given, the maximum size of a span locality can be expressed in terms of these parameters.

Some example networks are investigated, to gain an understanding of how RPL and d affect the magnitudes of S' and N'. In this analysis, it is assumed that quasi-planar (mesh telecommunications) networks have artificially uniform node degrees.

Consider the structures of Figure 4.3. These are constant node degree networks. Networks (a), (b), and (c) have node degrees of three, four and five respectively. The N' values within these networks can be upper-bounded as the number of nodes within RPL/2 hops (logical

distance) because one of the end-nodes of a failure will always be within that many hops of each node in the locality. The size of the locality can then be described geometrically as RPL increases. For example, in the d=4 network:

1 hop allows addition to the locality of the 4 adjacent nodes to the starting node.

2 hops also allows the 4 type-A nodes which are two hops directly up,left,down and right; and the 4 type-B nodes which are half way between the type-A nodes.

3 hops also allows the 4 type-A nodes three hops directly up,left,down and right; and the 8 type-B nodes which are 1/3 and 2/3 of the way between these new type-A nodes.

Continuing this process, it is noticed that the i-th hop adds 4·i nodes to the locality, except for the zero-th hop which adds a single node.

Therefore, $N' = 1 + 4 \cdot \sum_{i=1}^{RPL/2} i = \frac{RPL^2}{2} + RPL + 1 = O(\frac{4}{8} \cdot RPL^2)$

The degree 3 and 6 networks can be analyzed similarily to arrive at the N' values displayed in Table 4.2. These results show that N' is $O(d \cdot RPL^2)$.

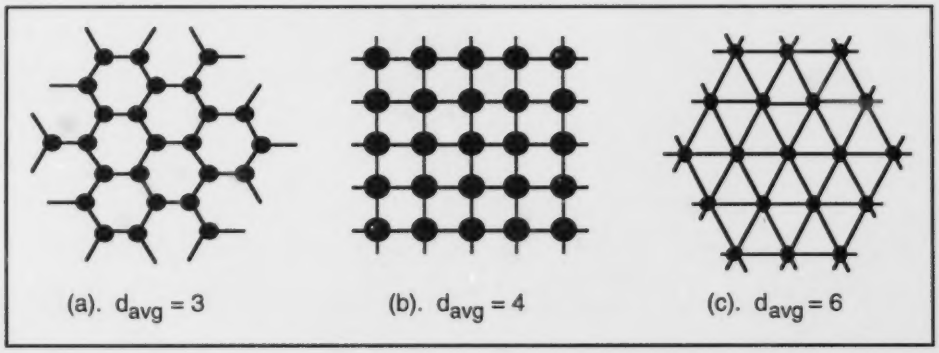


Figure 4.3 Constant Degree Network Topologies

Table 4.2 N' values in terms of RPL and d_{avg}

d_{avg}	N'
3	$N' = (3/8 \cdot RPL^2) + (3/4 \cdot RPL) = O(3/8 \cdot RPL^2)$
4	$N' = 4/8 \cdot RPL^2 + RPL + 1 = O(4/8 \cdot RPL^2)$
6	$N' = (6/8 \cdot RPL^2) + (6/4 \cdot RPL) + 1 = O(6/8 \cdot RPL^2)$
d	$N' = O(d/8 \cdot RPL^2) = O(d \cdot RPL^2)$

A relationship for S' can be derived from the relationship for N'. S' is equal to d·N'/2, but no larger than S. Therefore, S' is $O(\min(S, d^2 \cdot RPL^2/8))$.

When d and RPL are small in relation to the network size, as the network grows the asymptotic value of S' is $O(d^2 \cdot RPL^2)$.

In real networks the allowed values of d and RPL are limited (eg. $d \leq 5$ and $RPL \leq 10$ generally). Therefore, locality sizes have an upper bound and $O(N') = O(d_{max} \cdot RPL_{max}^2) = O(1)$ and $O(S') = O(d_{max}^2 \cdot RPL_{max}^2) = O(1)$. This fact will be used in the average-case analysis of SCP heuristics.

4.1.5 Common Data Structures

Efficient implementation of algorithms requires appropriately chosen data structures. This section introduces some of the data structures that are important to the efficient implementation of SCP algorithms.

4.1.5.1 Adjacency Matrices and Lists

Adjacency matrices and lists store information about a network topology by defining adjacent nodes. The SCP algorithms considered use both adjacency lists (adjL) and adjacency matrices (adjM).

The adjacency list contains a list of all adjacent nodes for each node in the network. Figure 4.4 illustrates a network topology and the corresponding adjacency list. The adjacency list has the minimum storage requirements of any structure used to store network topologies: $N_d = N \cdot d = O(N \cdot d)$, where N is the number of nodes and d is the average node degree. An algorithm that requires sequential identification of adjacent nodes can efficiently retrieve information from an adjacency list. Therefore, SCP algorithms use the adjacency list structure when (a) they must access the nodes sequentially by geography, or (b) they must minimize storage space.

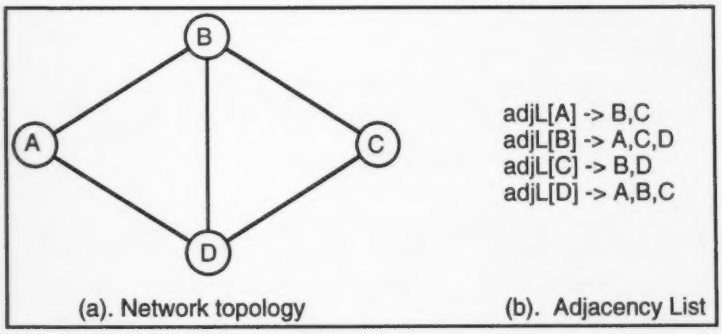


Figure 4.4 A Network Topology and the Corresponding Adjacency List

An adjacency matrix is a Boolean matrix that has a TRUE value for each element (i,j) for which Node i is adjacent to Node j. All other elements in the matrix are FALSE. For example, Figure 4.5 illustrates a network topology and the corresponding adjacency matrix. The storage requirement of the adjacency matrix is NxN bits of information, or $O(N^2)$, where N is the number

of nodes in the network. Initializing a data structure of $O(N^2)$ size takes a number of operations also of $O(N^2)$, because each element must be visited at least once. Therefore, this data structure is not only larger than the adjacency list, but it also requires more operations to initialize.

Adjacency matrices find their niche when an algorithm requires random information about the existence of spans. This information is directly accessible in an adjacency matrix, whereas it would require a brief search in an adjacency list. Thus, if the number of times that random access to span existence information is required exceeds the added complexity of initializing the adjacency matrix, the adjacency matrix is the data structure of choice.

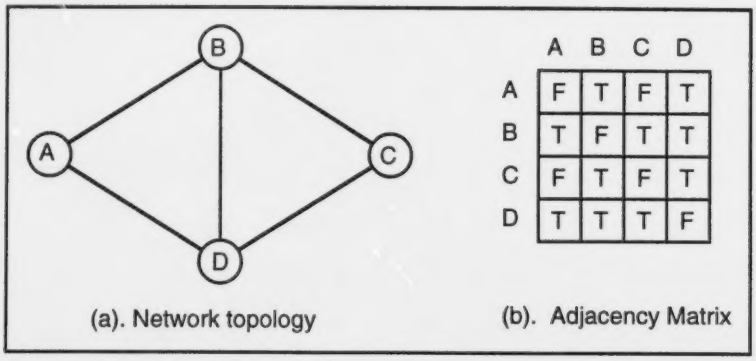


Figure 4.5 A Network Topology and the Corresponding Adjacency Matrix

The algorithms in this thesis have complexities exceeding that of either adjacency structure and, therefore, use whichever adjacency structure is most appropriate in each situation. Therefore, when an implementation can provide both adjacency structures without impacting asymptotic complexity. When the algorithm requires information about all of a node's immediate neighbors, the information is directly available through the adjacency list (rather than undertaking an O(N) search of the adjacency matrix). Conversely, when the algorithm requires random information about the existence of a span, the information can be accessed in a single operation from the adjacency matrix (rather than undertaking an O(d) search of the adjacency list).

In addition to providing adjacency information, the adjacency matrix can be easily adapted to contain a mapping from node pairs to span numbers in order to index vectors of span quantities. This function is effected by having the adjacency matrix elements hold span names rather than just Boolean values. Thus, a valid span name suggests a TRUE adjacency condition and also indicates the span name that joins those two nodes. A non-valid span name indicates a FALSE adjacency condition. With such mapping of node pairs to spans numbers, span information can then be contained in vectors of length S (the number of spans).

4.1.5.2 Binary Heaps

When performing operations on network topologies, an algorithm must sometimes maintain a list of nodes or spans in a dynamic priority list. For example, the Dijkstra shortest path algorithm requires a list of nodes maintained so that the closest node to any node contained in a set of selected nodes is quickly accessible. A dynamic priority list enables: (a) the quick retrieval of the highest priority element, (b) the addition of new elements and their priorities, and (c) the alteration of elements' priorities, as necessary. For example, in Dijkstra's shortest path algorithm, a simple list of nodes and their distances from the set of marked nodes is a dynamic priority list when sorted by distance. However, adding an element into the correct position in a simple sorted list requires O(n) operations, where n is the number of elements in the list. Binary heaps more efficiently manipulate dynamic priority lists. This section introduces the binary heap structure, describes its value to the SCP problem, and provides procedures that operate on the structure.

As depicted in Figure 4.6, the binary heap is a rooted tree-like structure. The root of the tree is element a. An element of a binary tree can have up to two children. For example, element b is a child of element a and a parent to its two children, elements d and e. The leaves of the tree are those elements that do not have children of their own. The depth of an element is its number of ancestors (parent, parent's parent, etc.). Thus, the root has a depth of 0, the root's children have a depth of 1, and so on. The level of an element is the maximum depth of the tree minus the depth of that element. Thus, elements with an equivalent number of ancestors share the same level and depth. The root is the only element at the maximum level. In a heap, the leaves are always arranged to be left justified on level 0 and on level 1 to the right of any element that has children, as depicted in the figure.

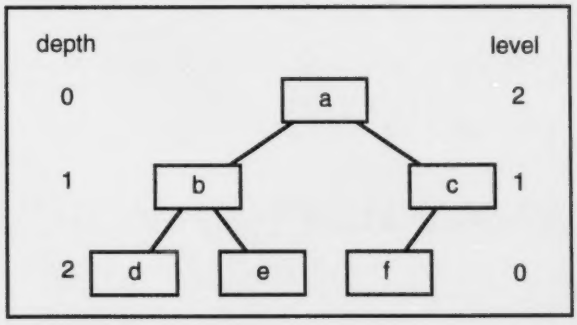


Figure 4.6 A Tree Structure

Figure 4.7 depicts a heap structure stored in a sequential block of memory. This structure does not require any explicit pointers to indicate which elements share a child-parent relationship because of a systematic mapping of elements to memory locations: A node at memory location i has children at memory locations 2i and 2i+1. Additional elements are inserted

at the first available memory location, which is always located at the end of the list. Thus, heaps have the same storage requirements as a simple list.

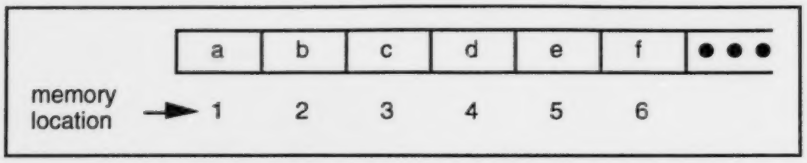


Figure 4.7 Heap Representation of Tree in Figure 4.6

When operating in a dynamic priority list, the binary heap structure requires fewer operations to maintain the list. Because the depth of the heap is $O(\log_2 N) = O(\log(N))$, the number of steps required in sorting operations that maintain the priority element at the root of the tree is also O(log(N)). Thus, using the binary heap instead of a sorted simple list reduces the time complexity of maintaining a dynamic priority list.

The common operators for maintaining a binary heap are the percolate and sift-down procedures of Figure 4.8. These procedures are as presented in [BrBr88], with the exception that our application gives priority to the minimum (rather than the maximum) valued element. For the implementation of dynamic priority lists, the procedures require an additional array for maintaining the location of each element within the binary heap and, thus, allowing fast access for making changes to the priority of an element that is already in the list. The implementations of Figure 4.8 do not include this additional array.

During insertion of new elements, the percolate() procedure is used to maintain the order of the elements of a heap. It takes an element at location i in the heap, h[1..n], and recursively exchanges that element with its parent until it is larger than the parent. When adding a new element to the heap, location i is location n.

sift-down() is a necessary procedure for efficiently deleting an element. After removing an element from a heap (usually the root or priority element), the element at the end of the heap is moved to the newly created gap and then sifted down to a new, sorted, position. The sift-down procedure is the opposite operation to the percolate procedure: an element is recursively compared to its children and exchanged with the smaller of its children until it is smaller than both children.

```
procedure percolate(h[1..n],i) {
    k <- i;
    repeat {
            j <- k;
            if (j > 1 AND h[j div 2] > h[k])
                    k <- j div 2;
            temp <- h[k];
            h[k] <- h[j];
            h[j] <- temp;
    } until (j = k);
}

procedure sift-down(h[1..n],i) {
    k <- i;
    repeat {
            j <- k;
            if (2j <= n AND h[2j] < h[k]) k <- 2j;
            if (2j < n AND h[2j+1] < h[k]) k <- 2j+1;
            temp <- h[k];
            h[k] <- h[j];
            h[j] <- temp;
    } until (j = k);
}
```

Figure 4.8 The Binary Heap percolate() and sift-down() Procedures

Figure 4.9 presents the insert() and delete-min() procedures, which use the percolate() and sift-down() procedures. insert() adds a new element to the heap and delete-min() removes the root and sorts the heap.

```
procedure delete-min(h[1..n]) {
    elem <- h[1];
    h[1] <- h[n];
    sift-down(h[1..n-1],1);
    return elem;
}
procedure insert(h[1..n],v) {
    h[n+1] <- v;
    percolate(h[1..n+1],n+1);
    return elem;
}
```

Figure 4.9 The Binary Heap delete-min() and insert() Procedures

4.1.6 Common Procedures

Procedures that perform operations on network topologies use the data structures presented in the previous section. Here, several of the more important procedures of the SCP algorithms are introduced.

4.1.6.1 Depth First Search (DFS)

Many algorithms that operate on graphs require an efficient method for visiting each of the network's nodes. The Depth First Search (DFS) serves this function.

A DFS is an algorithm that operates on a network and seeks to visit each node by traversing via the spans. After the algorithm completes, the group of spans that were traversed have the following characteristics: (a) they form a connected subnetwork, (b) there are no rings in the subnetwork (the subnetwork is one-connected), and (c) each node is included in the subnetwork. In mathematical terms, this subnetwork is a spanning tree: "spanning" refers to including all nodes; and "tree" means that no rings (or cycles, in mathematical terms) exist in this subnetwork.

The algorithm starts at an arbitrarily selected node and marks the node as visited. The algorithm then visits a node adjacent to the starting node (if any exist) and labels the new node as visited. The span between the starting node and the node visited next is part of the final spanning tree. The algorithm attempts to traverse the spans of the last node visited to find nodes that have not yet been visited. If it fails to find an unvisited node adjacent to the most recently visited node, it "falls back" to the second most recently visited node to continue its search, and so on. When a "fall back" results in the algorithm returning to the node at which it started, then either it has visited all nodes (in the case of a connected network) or it has visited all nodes in the connected subnetwork (in the case of a non-connected network). At this point, if any nodes have not been visited, the algorithm arbitrarily continues its search at one of these nodes. The algorithm proceeds until it has visited all nodes. The "depth first" part of the DFS algorithm's name refers to its preference of adding new spans to the longest path of spans first.

SCP heuristics can use the DFS to determine if a restoration path of defined path length limit exists for a specific span failure. The DFS is executed on a network composed of all the network's nodes and only the network's spans that have remaining spare capacity. If a restoration path exists, the DFS identifies a subnetwork containing both of the end-nodes of the failed span. If a restoration path does not exist, the DFS identifies two different subnetworks, each including one of the end-nodes of the failed span. It may also identify other subnetworks that contain neither of the end-nodes of the failed span. The benefit of the DFS in this situation is that the set of spans traversing the boundary between subnetworks represents a cutset through which restoration capacity is known to be unavailable. The ICH heuristic uses this important method of identifying cutsets.

Figure 4.10 contains the specific implementation of a DFS [Gibb85] used here.

```
1       main () {
2               F <- NONE;
                /* mark all nodes unvisited */
3               for all (theNode in Nodes) {
4                       DFI[theNode] <- FALSE;
5               }
6               while (DFI[u] = FALSE for some u in Nodes) {
7                       DFS(u);
8               }
9               output F;
10      }

11      procedure DFS (theNode) {
12              DFI[theNode] <- TRUE;
13              for all (theNode' in adjL(theNode)) {
14                      if (DFI[theNode'] = FALSE) {
15                              F <- F + [SpanName[theNode][theNode']];
16                              DFS(theNode');
17                      }
18              }
19      }
Legend:
                DFI[1..N]:              Boolean vector of visited nodes.
                F:                      spanning tree of spans traversed.
                Nodes:                  the set of all nodes.
                theNode, theNode':      specific nodes under consideration.
```

Figure 4.10 A Depth First Search

The complexity (in terms of the number of steps) of the DFS is O(max(S,N)), where S is the number of spans and N is the number of nodes. This complexity is present in several parts of the DFS algorithm. First, the DFS maintains the DFI array to ensure that each node is visited once, requiring O(N) steps (line 4 is initialization, line 6 searches the array). Also, the output of the algorithm is a spanning tree, which has O(N) spans. The algorithm performs the test at line 14 for each neighbor of each node, requiring 2S tests or O(S). Therefore, taking the maximum of these complexities results in O(max(S,N)) steps, as stated.

4.1.6.2 A Fast Implementation of Dijkstra's Algorithm

Dijkstra's algorithm operates on a network to find the shortest path from a single node to each other node in the network. Dijkstra's algorithm is used in this thesis to locate the shortest restoration paths by the SLPA heuristic algorithm for SCP.

Dijkstra's algorithm is similar in function to the DFS described in the previous section. Dijkstra's algorithm starts at a single node (the "source"), expands the subnetwork of visited nodes by incorporating one additional unvisited node at each iteration, and ends after visiting every node. The difference between Dijkstra's algorithm and DFS is the method of choosing

which node to incorporate at each iteration. Dijkstra's algorithm always adds the closest (either logical or geographical, depending on algorithm's use) unvisited node to the source. This is a "breadth-first" search, as opposed to the depth-first method of DFS.

Dijkstra's algorithm can best be understood with a brief example. In the network of Figure 4.11, the algorithm starts with Node A, the "source" node. The objective is to define the shortest path from the source node to each other node in the network. This example bases the path length on the physical lengths of spans, which are displayed beside each span in the figure. Initially, only the source node is marked as "visited." A node set "U" stores all visited nodes, and therefore set U initially only contains the source node. The next visited node is the closest adjacent node to the source node, which in this case is Node B. Set U now contains two members (nodes A and B). The algorithm next considers all nodes adjacent to a member of set U, namely nodes C, D and E. Node C is 6 units away from the source, and Node D is 5 units away from the source (2 units to Node B plus 3 units from B to A). Therefore, the shorter distance is to Node D and the algorithm visits Node D next and adds it to U. Finally, Node C, then Node E are similarly added to U. As was the case in DFS, the spans traversed in visiting the nodes form a spanning tree. Figure 4.11 displays the spanning tree produced in the example in bold lines. The spanning tree defines the shortest routes from each node to the source node. To assist in identifying the shortest path between the source and each other node in post processing, the algorithm stores the span over which each node was first visited. The predecessor node array (pred[1..N]) contains the traversed span information as the adjacent node's name. For example, the algorithm first visited Node D via Node B that was the closest adjacent node (in U) and, therefore, the predecessor of Node D is Node B (pred[D] = B). Then, the predecessor trail contains the shortest path from Node D to the source node (Node A): pred[pred[D]] = pred[B] = A.

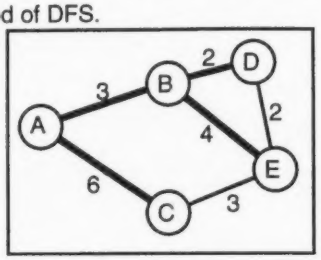


Figure 4.11 Shortest Path From A to Each Other Node

SCP heuristics use Dijkstra's algorithm iteratively to identify the k successively shortest restoration paths for a span failure, as first introduced in [Grov89]. This use of Dijkstra's algorithm is accommodated by designating the end-nodes of the failed span as the source node and the target node (random orientation). Dijkstra's algorithm operates on a subnetwork including all the network's nodes and the network's spare capacity. After executing Dijkstra's algorithm, the trail of predecessors leading from the target to the source identifies the minimum length route that contains a spare path(s).

Figure 4.12 contains a general implementation of Dijkstra's algorithm. Although it is based on [Gibb85], this implementation incorporates some of the designations used in the more advanced algorithm to be presented later in this section.

```
Procedure Dijkstra (source) {
1   for all (theNode in Nodes) {
2           minDist[theNode] <- INF;
    }
3   for all (theNode in adjL[source]) {
4           minDist[theNode] <- dist[source,theNode];
5           pred[theNode] <- source;
    }
6   minDist[source] <- 0;
7   pred[source] <- source;
8   U <- [source];
9   while (U <> Nodes) {
10          find (any minNode in [Nodes\U]) such that for (all aNode in [Nodes\U]) {
11                  minDist[minNode] <= minDist[aNode];
            }
12          U <- U + [minNode];
13          for all (theNode in adjL[minNode]) {
14                  extradist <- dist[minNode,theNode];
15                  if ((theNode not in U) AND
                            (minDist[minNode] + extraDist < minDist[theNode])) {
16                          minDist[theNode] <- minDist[minNode] + extraDist;
17                          pred[theNode] <- minNode;
                    }
            }
    }
}
Legend:
            Nodes:              the set of all nodes.
            U:                  the set of nodes which have been visited.
            minDist[1..N]:      the shortest known distance to each node from the source.
            pred[1..N]:         the next span in the predecessor trail back to the source.
            dist[node1][node2]: the length of the span between node1 and node2.
                                (set to 1 or 0 if logical distances are used).
```

Figure 4.12 An Implementation of Dijkstra's Shortest Path Algorithm

The complexity of Dijkstra's algorithm depends upon the data structures. The loop starting at line 9 iterates N times, where N is the number of nodes in the network. For each of these N iterations, the search at line 10 executes once. If the sets U and Nodes are implemented as simple lists, this inner loop also requires O(N) steps. The loop starting at line 13 only iterates O(d) = O(N) times. Therefore, a traditional implementation of Dijkstra's algorithm executes in $O(N^2)$ time. The implementation used in this thesis reduces this complexity by selecting more appropriate data structures.

[MaGr89] describes an enhanced Dijkstra implementation specifically optimized for finding sets of k restoration paths in quasi-planar networks. This algorithm, appearing in Figure 4.13 stops executing as soon as the target node is visited. It also stores the unvisited nodes in a binary heap and, therefore, can perform operations on sets of nodes in O(log(N)) steps rather than O(N) steps as in a list implementation.

This improved implementation has an overall complexity (again in terms of the number of operations) of O(N·log(N)) steps. As before, the loop starting at line 15 iterates O(N) times, but now heap_deletemin() (line 16) uses a binary heap and can locate the closest node and update the data structure in O(log(N)) steps. Thus, iterated by the loop of line 15, line 16 has O(N·log(N)) steps, as stated. Lines 23 and 24 will generally only be executed once for each node, as nodes are first marked. Thus, the algorithm executes these lines a total of O(N) times, independently of the surrounding loops. With an internal complexity of O(log(N)), the heap_decrease() procedure of line 23 also consumes an amount of time in O(N·log(N)), leaving the complexity as originally stated.

The metaDijkstra() algorithm executes Dijkstra k consecutive times to locate k-shortest link disjoint restoration paths and therefore restore a failed span with k working links. Thus, using the optimized dijkstra() of Figure 4.13 as a kernel, metaDijkstra() locates the k-shortest paths in O(k·N·log(N)) steps. The metaDijkstra() implementation used here (Figure 4.14) improves on this complexity by accepting all paths that a route identified by dijkstra() can support. Also, when working in a logical distance mode, diverse routes with equivalent length commonly exist. As a side benefit, each dijkstra() call also returns information about some alternate routes that may contain restoration capacity. metaDijkstra() explores these alternate routes, after accepting the primary route, by picking up the predecessor trail at nodes adjacent to the target, as follows: "any (theNode in adjL[target]) such that (minDist[theNode] + dist[spanName[theNode][target]]) = minDist[target]."

```
procedure dijkstra(NODE: source, target, minDist[1..N], pred[1..N];  SPAN:  dist[1..S]) {
1           for all (theNode in Nodes) {
2                   mark[theNode] <- FALSE;
3                   pred[theNode] <- NONE;
  }
4           mark[source] <- TRUE;
5           for all (theNode in [Nodes - source]) {
6                   if (theNode = target OR (adjM[source][theNode] = FALSE))
7                           heap_insert(INFINITY, theNode);
8                   else    heap_insert(dist[spanName[source][theNode]],theNode);
9                   minDist[theNode] <- INFINITY;
  }
10          for all (theNode in adjL[source]) pred[theNode] <- source;
11          pred[target] <- NONE;
12          pred[source] <- source;

13          i <- 0;
14          found <- FALSE;
15          while (found = FALSE AND i < N) {

16                  minNode <- heap_deletemin(minNodeDist);
17                  minDist[minNode] <- minNodeDist;
18                  mark[minNode] <- TRUE;
19                  if (minNode = target) found <- TRUE;

20                  for all (theNode in adjL[minNode]) {
21                          extraDist <- dist[spanName[theNode][minNode]];
22                          if (mark[theNode] = FALSE AND
                            (minNodeDist + extraDist < heapDist[theNode])) {
23                                  heap_decrease(theNode, minNodeDist + extraDist);
24                                  pred[theNode] <- minNode;
                    }
            }

25                  i <- i + 1;
  }
}
Legend:
            mark[1..N]:                             Boolean vector of visited nodes.
            heap_decrease & heap_deletemin:         the decrease() and deletemin()
                                                    procedures introduced in Section 4.1 5.2.
```

Figure 4.13 A Fast Implementation of Dijkstra's Shortest Path Algorithm

```
procedure metaDijkstra(NODES:  source, target) {
        for all (theSpan in Spans) {
                spares[theSpan] <- conSpare[theSpan];
                dist[theSpan] <- conDist[theSpan];
                if (spares[theSpan] <= 0)
                        dist[theSpan] <- INFINITY;
        }

        numPaths <- 0;
        stop <- FALSE;
        while (stop = FALSE) {

                dijkstra(source, target, minDist[1..N], pred[1..N], dist[1..S]);

                if (minD[target] < INFINITY)
                for all (theNode in AdjL[target]) {
                        if (pred[theNode] != NONE AND theNode != source AND
                                minDist[theNode] = minDist[target] -
                                dist[spanName[theNode][target]]) {

                                j <- theNode;
                                maxDepth <- conspare[spanName[target][j]];
                                while (j != source AND maxDepth > 0) {
                                        i <- spanName[pred[j]][j];
                                        if (maxDepth > conSpare[i])
                                                maxDepth <- conSpare[i];
                                        j <- pred[j];
                                }

                                numPaths <- numPaths + maxDepth;

                                if (maxDepth > 0) {
                                        j <- target;
                                        pred[target] <- theNode;
                                        while (j != source) {
                                                i <- spanName[pred[j]][j];
                                                conspare[i] -= maxDepth;
                                                if (conSpare[i] <= 0)
                                                        dist[index] <- INFINITY;
                                                j <- pred[j];
                                        }
                                }
                        }
                } else stop <- TRUE;
        }
        return (numPaths);
}
```

Figure 4.14 The metaDijkstra Procedure

4.2 Network Flows

When designing a mesh restorable network, spare paths over alternate routes protect each span's working capacity. The number of spare paths required to protect a span is equivalent to the number of working links on that span. This section explores the possible methods that a flow-based restorability test may use to assess the number of alternate paths available.

The number of paths that exist between a source node and a target node in a network is a capacity flow. This "flow" is analogous to the amount of water that can flow through a system of pipes from an intake (source) to an outlet (target). The minimum cross-sectional area of pipe dictates the maximum amount of water that can flow through the system. Similarly, the "max-flow" through a network is the maximum possible number of paths (or capacity) available between the source node and the target node. Thus, a "max-flow" calculation can be used to assess the restorability of a network because it returns the greatest number of feasible restoration paths.

However, to realize the capacity that a max-flow calculation identifies, the paths must be chosen to ensure that no critical capacity is wasted. For example, in the network of Figure 4.15, the maximum flow between nodes 1 and 4 is 2 paths; this maximum capacity is realized by selecting paths 1-5-6-3-4 and 1-2-7-8-4. However, if a restoration algorithm selects the shortest path (path 1-2-3-4), it wastes some capacity because no second path is available.

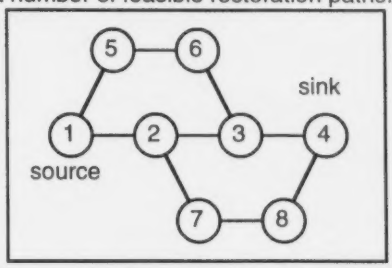


Figure 4.15 Example Where k-shortest Paths Differs From Max-flow

Max-flow capacity can only be achieved with particular path selections. The complexity of a path selection technique that realizes max-flow capacity is at least as large as the complexity of an algorithm which identifies the max-flow in a network. The lowest complexity max-flow algorithm requires $O(N^3)$ steps [Gibb85]. (It does not benefit in the same way from data structure selection as Dijkstra did when achieving a reduction of one of the N dependences to log(N).) However, metaDijkstra (Section 4.1.6.2) finds the k-shortest link-disjoint paths in $O(k \cdot N \cdot \log(N))$ steps. Therefore, because speed is of the essence, restoration algorithms for operating on mesh restorable networks generally find the k-shortest restoration paths rather than the max-flow restoration paths. A k-shortest path identification algorithm selects the shortest path first, then the second shortest path that is link-disjoint from the first, and so on, until the k-shortest paths are identified. The example of Figure 4.15 demonstrates that this k-shortest link-disjoint path selection does not necessarily result in a max-flow. However, [DuGr89] showed that a k-shortest paths flow is only a few percentages lower than max-flow in a large number of randomly generated telecommunications networks. The benefits of metaDijkstra are therefore several in

practice: the path set is identified, not just the feasible flow quantity and the flow of the k-shortest path set is very nearly equal to the ideal max-flow quantity. Max-flow however is $O(n^3)$ and leaves the identification of the path-set that realizes that flow as a separate problem.

The remainder of this thesis therefore uses k-shortest paths flow as the criterion of restorability. However, this too is only an approximation of the flow that a restoration algorithm would obtain when operating on the real network, because the "k-shortest link-disjoint path flow" depends upon the order of path selection when executing with logical link lengths.

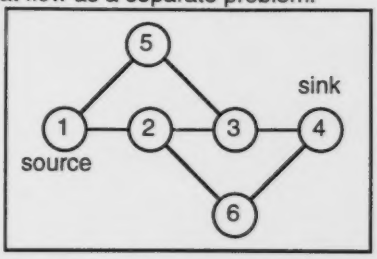


Figure 4.16 Example Where k-shortest Paths "Flow" Depends Upon Selection Order

Therefore, the only truly accurate measure of restorability is an execution of the restoration algorithm itself. For example, in the network of Figure 4.16, the k-shortest paths flow can be only one if the algorithm selects path 1-2-3-4 first, or it can be two paths, if the algorithm selects path 1-5-3-4 first. Both first choices are equally valid because logical routing lengths are used. Within this paper, results quantify the disparity in flow calculations that can result from different orders of path selection within k-shortest path algorithms.

4.3 Formulation of the Optimal Spare Capacity Placement Problem

The main issue addressed by this thesis is the problem of spare capacity placement (SCP) in a mesh restorable network so that two objectives are simultaneously achieved: (a) To protect every span against failure with spare paths on alternate routes, where any restorability level may be chosen, and (b) to minimize the total spare capacity.

The most general statement of the SCP problem is:

$$\text{minimize} \sum_{j=1}^{S} s_j \tag{4.1}$$

such that

$$R_n = L, w_i \geq 0, s_i \geq 0 \; \forall i \in (0...S)$$

$$0 \leq L \leq 1$$

S is the number of spans in the network. s_i and w_i are the number of spare and working links respectively on span i. L is the desired restorability level input by the user.

In practice, the constraint $R_n = L$ is more usefully written as a vector of constraint relationships pertaining to the restorability of each span individually:

$$k_i \geq L \cdot w_i \; \forall i \in (0...S) \tag{4.2}$$

k_i is the number of restoration paths available for span i.

The min-cut max-flow theorem [FoFu56] states that the minimum of the flows through all possible cutsets of the network separating nodes s and t is equivalent to the maximum flow between s and t. The significance of this theorem can be illustrated by returning to the analogy of

networks of water pipes introduced in the previous section. The network bottle-neck constrains the maximum flow of water between the intake (s) and the sink (t). Here, the bottle-neck is the cross-section of the network containing the smallest area of pipe. In a similar manner, the cutset (or cross-section of spans) of the transport network which contains the least capacity determines the maximum capacity flow possible between the nodes s and t.

The min-cut max-flow theorem provides a method for converting restoration requirements of (3.2) into inequalities that an integer program (IP) can then solve. For example, Figure 4.17 shows the series of cutsets that constrain the flow between nodes A and B, labeled as C_1, C_2, C_3 and C_4. An inequality is formed for each cutset that guarantees that the flow through the cut is at least as large as the working capacity on span A-B. For example, the inequality for cutset C_1 of Span A-B is as follows: $s_{AC} + s_{AB} <= w_{AB}$, where s_{ij} is the spare capacity on the span between nodes i and j, and w_{AB} is the working capacity on span A-B.

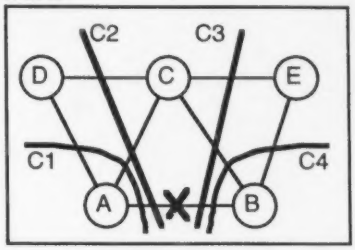


Figure 4.17 Cutsets Which Guarantee Restoration Flow for Span A-B

All the possible cutset constraints can be expressed as an IP, where the IP must minimize the total spare capacity. The IP performs a gradient search over the linear constraints to converge to an optimum solution. Specifically, the integer program is:

$$\text{minimize} \sum_{j=1}^{S} s_j \tag{4.3}$$

such that: $\mathbf{C} \bullet \mathbf{s} \geq \mathbf{w}' \cdot L$ (4.4)

where: $s_j, w_j \geq 0 \; \forall j \in (0...S)$ (4.5)

Equation is depicted in an expanded form in Figure 4.18. **C** is the N_CxS cutset matrix of the network, where N_C is the number of cutsets in the network, and S is the number of spans in the network. Each row of **C** specifies one cutset, and the spans included in the cutset are represented by 1's in the appropriate column. **s** is the length S vector of spare capacities to be determined by the program. **w** is the length S vector of working capacities input as constraints to the program. **w'** is a length N_C working capacity vector, where each element of **w'** is an element of **w**. Each w_i appears in **w'** as many times as there are cutsets for constraining the restoration capacity of span i. w_i appears as the j-th element of **w'** if the j-th row of **C** represents a cutset constraint for ensuring restoration flow for w_i.

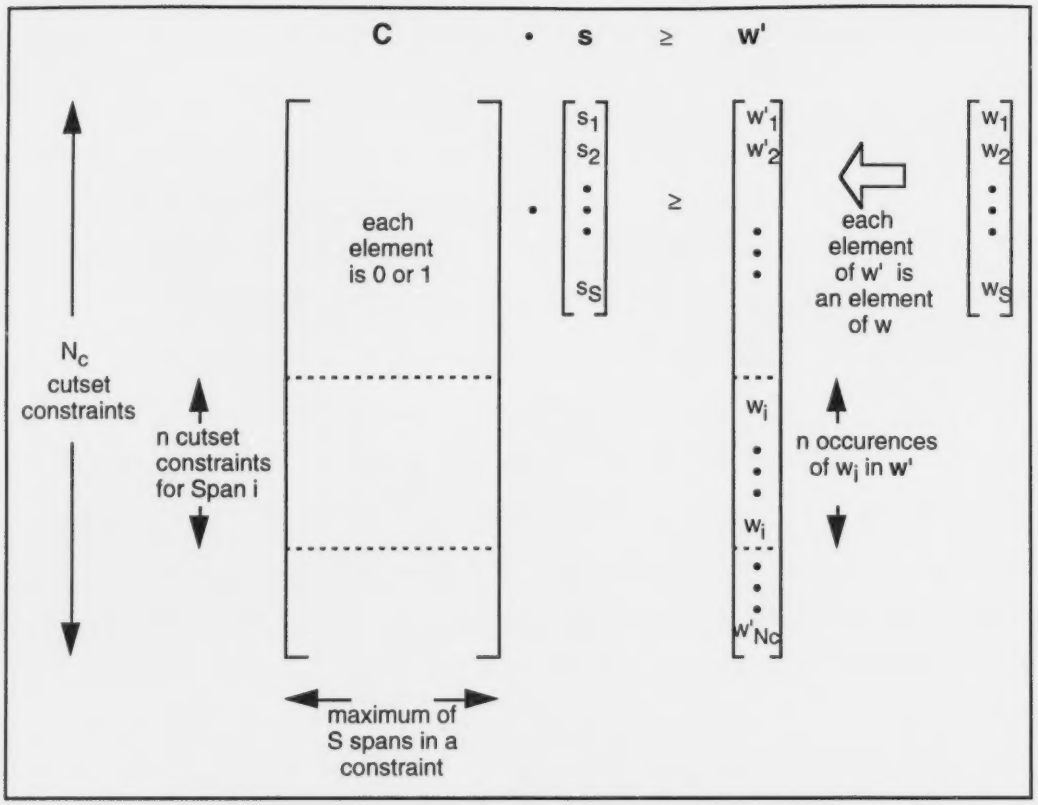


Figure 4.18 A Description of the Elements of Equation (4.4)

Called the direct cutsets algorithm (DCA), this method of SCP is easily implemented. DCA can use a real-valued Linear Program (LP), which is less computationally intensive than an IP [Schr86]. LP's are solvable with commercially available software packages (Mathematica™ in the current research). But, the number of cutsets required to fully specify the LP increases exponentially with network size (Chapter 5). Therefore, while easily implemented, the large number of constraints required to meaningfully formulate a real problem is itself a computational problem that makes DCA infeasible for even moderately sized networks. The iterative cutsets heuristic (ICH), described in Chapter 6, uses the cutset approach of DCA, but seeks to find a solution with a subset of the full constraints through iteration.

5 The Complexity of Spare Capacity Placement

This chapter provides a proof of the aforestated assertion that the SCP problem is NP-hard -- a member of the class of "hard" problems.

Section 5.1 provides required background knowledge in complexity theory. The discussion includes an introduction to the NP-complete and NP-hard classification of problems. Following the theoretical proof of NP-hardness of SCP in Section 5.2, Section 5.3 documents experimental results which confirm SCP's complexity. These experiments investigate the complexity present in the Direct Cutsets Algorithm of Chapter 4.

5.1 Introduction to Complexity Theory

This introduction to complexity theory follows a similar presentation in [BrBr88].

P is a class of decision problems that an algorithm can solve in polynomial time. For brevity, here **P** also designates algorithms that can execute in time on the order of a polynomial function of their input sizes. In algorithm design, a problem cannot be efficiently solved without an algorithm in **P**.

Depending on the exponent of the polynomial which describes the algorithm complexity, even algorithms in **P** can be inefficient (indeed, exponents greater than 3 are often considered inefficient for many applications). However, algorithms not in **P** are definitely inefficient (for example, exponential, power or factorial cost functions).

NP is the class of problems that a non-deterministic algorithm can solve in polynomial time. A non-deterministic algorithm has two phases, the guessing phase and the testing phase. In the guessing phase, the algorithm generates a solution to the problem. In the testing phase, it tests this solution for correctness. Algorithms that solve problems in **NP** have testing phases that operate in polynomial time and space. Thus, with a good initial guess, the algorithm generates a solution in polynomial time. The **NP** class includes **P** problems, by definition.

[Cook71] identified a class of problems, within **NP** labeled NP-complete, that are all equivalent and "hard". Many problems which have been pondered for centuries, including the Chinese postman problem, the traveling salesman problem, and the knapsack problem, are NP-complete. With the magnitude of effort expended trying to discover efficient algorithms for these and other NP-complete problems, it is reasonable to assume that efficient algorithms do not exist to solve NP-complete problems. Therefore, it would be wasteful to spend time trying to discover an efficient exact solution for optimum SCP if it is NP-hard (the set of problems at least as hard as NP-complete problems).

A proof of equivalence between problems includes reducing one problem to the other through operations in **P** and vice versa. This reduction includes three steps, as depicted in Figure 5.1. First, convert the input to Algorithm B (x) to provide input to Algorithm A through the function

S(x). Then algorithm A operates on the data to form a solution, y. Finally, convert the output of Algorithm A to the output expected by Algorithm B through the function T(y). In essence, this process uses Algorithm A and the conversions produced by S(x) and T(y) in order to implement Algorithm B.

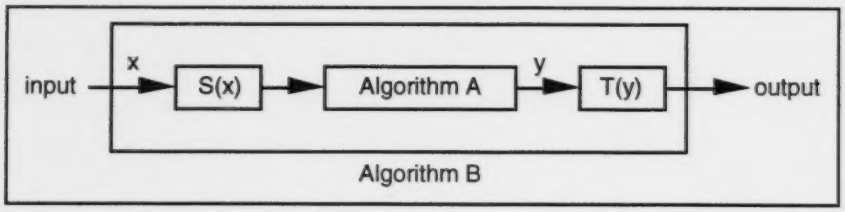


Figure 5.1 An Algorithm B can be Reduced to an Algorithm A by the Transformations S and T

If both functions S(x) and T(y) execute in polynomial time Algorithm B is polynomially reducible to Algorithm A, written as:

$$\text{Algorithm B} <=^{P} \text{Algorithm A} \qquad (5.1)$$

If a polynomial time algorithm for A is known, then (5.1) asserts that an algorithm for problem B exists which is also in **P**. The next section will use this form of reduction to prove that the SCP problem is hard by reducing a known hard problem to SCP.

5.2 Complexity Proof of Spare Capacity Placement

The existence of a Hamiltonian Cycle in a graph is one of the classic NP-complete problems. If a polynomial time solution exists for SCP, then by the reduction which follows, a polynomial time solution exists for Hamiltonian Cycle. Because no such solution is known for Hamiltonian Cycle or any of the other related NP-complete problems, it is doubtful that a polynomial time solution exists for SCP.

A Hamiltonian Cycle is a ring (cycle) which visits every node exactly once. In Figure 5.2(a), the network does not have a Hamiltonian Cycle, because the cycle must pass through Node A more than once in visiting every other node. In Figure 5.2(b), the graph has a Hamiltonian Cycle, traced in bold spans. Clearly, the sketches of Figure 5.2 imply a relationship between restorability and the existence of a Hamiltonian Cycle, because the network which did not contain a Hamiltonian Cycle also included a node which, if failed, would disconnect the network.

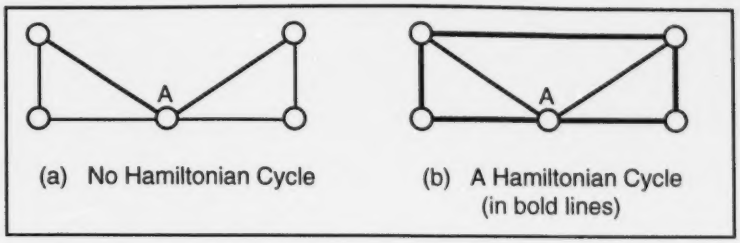


Figure 5.2 Hamiltonian Cycles

5.2.1 Reduction of a Hamiltonian Cycle to Spare Capacity Placement in a Network With One Working Link on Each Span

In a network comprised of spans that each carry a single working link, optimum SCP will place a single spare link on each span of a Hamiltonian Cycle and no spare links on the other spans. A proof follows.

A network with a single working link on each span requires at least two adjacent spare links per node. If only one spare link is adjacent to a node, a failure of the span containing the spare link leaves no route to restore the failed working link. Such a situation is illustrated in the network of Figure 5.3.

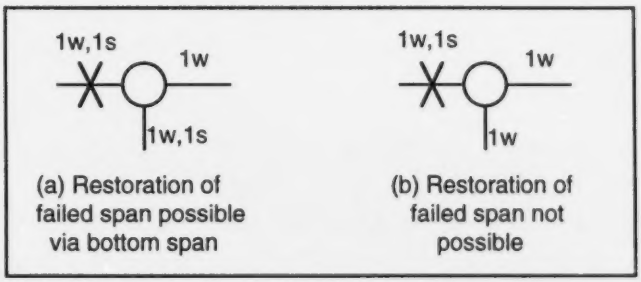


Figure 5.3 Demonstration of Requirement of 2 Spare Links per Node

If, however, the network contains a Hamiltonian Cycle, placement of one spare link on each span of the cycle provides full restorability because the cycle connects each pair of nodes via two span-disjoint spare paths. Therefore, an optimum SCP in a network that contains a Hamiltonian Cycle has an average of no more than two adjacent spare links per node.

Because this network requires at least two adjacent spare links, and it also requires at most two adjacent spare links, optimum SCP in this specific network requires exactly two adjacent spare links per node.

The only configuration using two adjacent spare links per node which satisfies optimum SCP is a Hamiltonian Cycle of spare links. In order for a span failure containing a spare link to be restorable, there must be an alternate path of spare links through the network to provide restoration of the failed working link; see Figure 5.4. Thus, the spare link on the failed span, together with the spare links on the restoration route, make a cycle or ring. Each of the spans which has a spare link must be part of a restoration ring like this one.

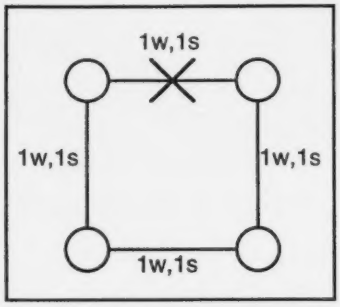


Figure 5.4 Formation of a Cycle by a Restoration Route

If more than one of these restoration rings appears in an optimum SCP, they cannot contain a common node, because that would imply a node with greater than 2 adjacent spare links -- a contradiction. However, if the restoration rings do not share a node, the working circuits on spans connecting a node on one restoration ring to a node on another restoration ring cannot be restored at all; see Figure 5.5.

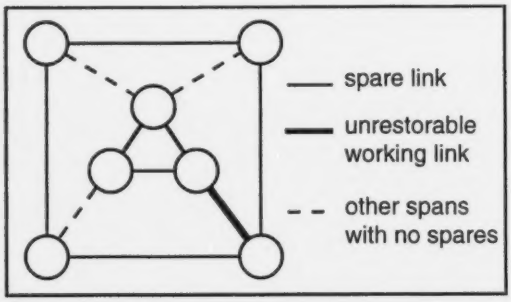


Figure 5.5 An Unrestorable Span

Therefore, if a Hamiltonian Cycle exists in the network with one working link per span, the spans with spare capacity in an optimum SCP form a Hamiltonian Cycle.

The Hamiltonian Cycle existence problem and all other NP-complete problems are formulated as decision problems. The preceding reduction not only solved the NP-complete Hamiltonian Cycle Existence problem, but also solved the even harder problem of identifying a Hamiltonian Cycle. The special case SCP problem solved above (1 working link per span) can be formulated as a decision problem for categorizing the topology: Can a single restoration ring be placed so that all spans are protected?

5.2.2 Conversions of the Input and Output Data

The first conversion, S(x), takes the network description, x, which was input to the Hamiltonian Cycle problem and generates input data for the SCP problem. This conversion is trivial because both Hamiltonian Cycle and SCP require a network description, x, as input. SCP also requires a vector **w** of working capacities, in this case specifying one link per span. The

creation of this vector is linear in S, the number of spans. The SCP algorithm can then be used to place spare capacity for restoration of the input network.

The output from SCP can be converted to the solution of the Hamiltonian Cycle existence problem -- the operation T(y) -- by summing the length S vector of spare capacities. If the total number of spare links is equivalent to the number of nodes in the network, and the network is 100% restorable, then a Hamiltonian Cycle exists; otherwise it does not. This conversion is also linear, O(S). Therefore, Hamiltonian Cycle $<=^{P}$ SCP.

5.2.3 Conclusion

Hamiltonian Cycle existence is polynomially reducible to optimum SCP, for the specific class of networks with one working link per span. If there was a polynomial-time algorithm to solve optimum SCP, then there would be a corresponding polynomial-time algorithm to solve Hamiltonian Cycle existence. However, no algorithms with less than exponential complexity are known for Hamiltonian Cycle and other NP-complete problems; therefore, the SCP problem is considered NP-hard and an efficient algorithm for SCP is unlikely to exist. This justifies the approach of finding a polynomial-time heuristic which solves the SCP problem in a nearly optimal way.

5.3 Complexity of the Direct Approach to Optimum SCP

The direct cutsets algorithm (DCA), introduced in Chapter 3, is an algorithm for calculating the optimum SCP. DCA is easy to implement; however, it has exponential complexity, as would any algorithm which exactly solves optimum SCP. This section presents experimental results which confirm that DCA cannot function in **P**. These experiments also provide insight for implementation of the Iterative Cutsets Heuristic (ICH), a DCA-style heuristic which operates with a limited number of cutsets. As a prelude to introduction of ICH in Chapter 6, this section discusses some of the issues involved in selecting a set of cutsets for LP constraints.

5.3.1 Number of Cutsets in Direct Cutsets Algorithm

The number of cutsets that exist in a network with N nodes grows exponentially with N. In a fully connected network, the number of cutsets is $N_C = N_{Cmax} = 2^{N-1}-1 = O(2^N)$. A cutset can be thought of as the set of spans through which a separation line is established between two distinct sub-networks of nodes. Each node can be assigned to either sub-network, and each assignment corresponds to a cutset in a fully connected network. The process of assigning nodes to one sub-network or the other is analogous to a binary digit; 2^N combinations exist for assigning nodes between sub-networks. Two additional restrictions reduce these 2^N combinations to $2^{N-1}-1$ cutsets. First, half of the sub-networks are not distinct, each assignment is repeated twice. Second, each sub-network must have at least one node, which eliminates a

scenario in which all nodes are in the same set. Thus, the number of constraints required to fully specify the problem is generally $O(2^N)$.

However, in quasi-planar networks N_c is less than N_{cmax} because the average node degree is lower and all cuts are not cutsets. But, clearly even in quasi-planar networks, N_c increases exponentially with network size. However, even if N_c does increase polynomially with network size in some restricted topologies, no method is known of identifying these cutsets in less than $O(2^N)$ steps. The reason identification of cutsets is a hard problem is the topic of the next section. Later in this chapter, results address the current question (numbers of cutsets) by empirically illustrating that N_c does indeed increase exponentially with network size.

5.3.2 Identifying Cutsets in Direct Cutsets Algorithm

The level of difficulty in identifying cutsets can be illustrated by considering the complexity of the following direct algorithm:

1. Divide the nodes into two groups, network component 1 and network component 2.
2. Test for (graph) connectivity of each component. (Only spans connecting two nodes in the same component can be used in the connectivity check.)
3. If both components are connected, this separation of nodes represents a cutset.
4. If either of the components is not connected, then this partitioning is not a cutset.

The algorithm for identifying the cutset constraints requires a connectivity check of each of the 2^{N-1} divisions of nodes into groups and, therefore, is $O(2^N)$. This method is not in **P**.

Another common algorithm for identifying cutsets starts from a spanning tree [Gibb85]. The nodes can be divided into two connected subnetworks by removing any span in the spanning tree. All of these divisions of nodes, from a single spanning tree, define fundamental cutsets for the network, as depicted in Figure 5.6. All other cutsets can be identified by circular additions of the fundamental cutsets. Circular additions refer the the addition of the sets of spans identifying cutsets where spans which appear even times do not appear in the new set. This leads to the same number of connectivity checks as the previous method: There are N-1 fundamental cutsets. An algorithm considers all combinations of these fundamental cutsets, resulting in $O(2^N)$ connectivity checks.

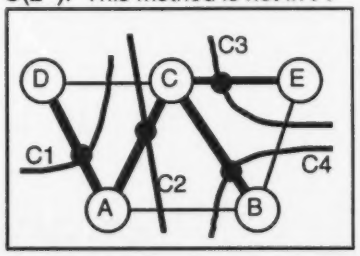


Figure 5.6 The Fundamental Cutsets Defined by the Spanning Tree (in Bold)

5.3.3 Methods of Generating Subsets of DCA Constraints

The number of cutsets which fully specify the DCA, $O(2^N)$, exceeds feasible limits of current computing platforms for even the smallest of the study networks. This section

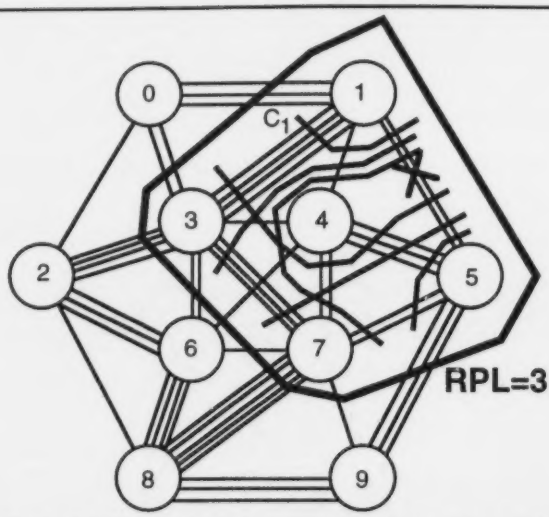
investigates methods for limiting the number of cutsets required for LP specification, and establishes a foundation for a cutset-based heuristic.

Two principles have been developed for producing a meaningful partial set of cutsets are restricting the restoration path length (RPL) and generating only the "bottleneck" constraints.

Using span localities (Section 4.1.4), specifying all of the distinct cutsets within the locality guarantees the restorability of a span in a DCA formulation. These cutsets are truncated versions of the larger cutsets which would be generated for the DCA, forcing the provision of capacity for restoration paths through a limited topological area of the network as demonstrated in Figure 5.7. In this case, there are an exponential number of cutsets with respect to the size of the locality, $O(2^{N'})$, but not with respect to the overall network size. In a network with a maximum node degree of d, using limited RPL, the number of cutsets which fully specify DCA is $O(2^{d \cdot RPL^2}) = O(1)$ (with respect to the network size, N). However, only limited values of d and RPL can be considered in practice. This method is described as over-constrained because the result may not be an optimal solution, due to imposed RPL limitations.

Figure 5.8 presents the increase in the number of truncated cutsets (N_{tc}) which results from increasing the size of the locality, via the RPL limit. The investigation includes several quasi-planar networks, generated by the methods described in Chapter 4. Each network had an average node degree of 3.7. The analysis above predicts $N_{tc} = O(N \cdot 2^{min(N',N)})$ and $N_{tc} = O(N \cdot 2^{min(d \cdot RPL^2,N)})$. Therefore, for the constant d and N curves of Figure 5.8, $N_{tc} = O(1)$ if $d \cdot RPL^2 > N$, and $N_{tc} = O(2^{RPL^2})$ otherwise. To test this, Figure 5.8 plots N_{tc} versus RPL^2 (not RPL). In the figure, $N_{tc} = E(1)$ for the N=9 network. Also, for large RPL values in the N=9 through 39 networks, the curves are leveling out to constant values (E(1)). However, in networks larger than N=39, N_{tc} is at least $E(2^{RPL})$ and in the N=94 node network, $N_{tc}=E(2^{RPL^2})$. Therefore, for the RPL limits considered, locality constraints can only restrict N_{tc} to less than exponential values with respect to RPL in networks smaller than 50 nodes.

The data in Figure 5.8 appears in a different format in Figure 5.9. Here, N is the independent variable, and the effect of RPL is observed as the networks increase in size. With a constant d and RPL, $N_{tc} = O(N)$ if $d \cdot RPL^2 < N$ and $N_{tc} = O(N \cdot 2^N)$ otherwise. Exponential increase in N_{tc} is observed for RPL≥6. In order to benefit from the E(N) increase of N_{tc} in the networks up to 100 nodes studied here, RPL must be limited to 4 or 5. However, it has been experimentally observed that the available LP cannot execute on more than 500 constraints in reasonable time (one day). Thus, Figure 5.9 shows that feasible use of truncated cutsets for 100-node networks requires restriction of the RPL to three. Therefore, this principle alone cannot reduce the number of cutsets to reasonable levels without over-constraining the restoration paths to an RPL of 3.



The locality restriction reduces the number of spans in a cutset and the total number of cutsets. For example, cutset C_1 provides the constraint:

$s_{1\text{-}3}+s_{1\text{-}4} \geq w_{1\text{-}5}$

This single constraint replaces all of the following constraints which were required before limiting the restoration path length:

$s_{1\text{-}3}+s_{1\text{-}4}+s_{0\text{-}1} \geq w_{1\text{-}5}$

$s_{1\text{-}3}+s_{1\text{-}4}+s_{0\text{-}1}+s_{0\text{-}3}+s_{0\text{-}2} \geq w_{1\text{-}5}$

$s_{1\text{-}3}+s_{1\text{-}4}+s_{0\text{-}1}+s_{0\text{-}3}+s_{2\text{-}3}+s_{2\text{-}6}+s_{2\text{-}8} \geq w_{1\text{-}5}$

$s_{1\text{-}3}+s_{1\text{-}4}+s_{0\text{-}1}+s_{0\text{-}3}+s_{2\text{-}3}+s_{2\text{-}6}+s_{6\text{-}8}+s_{8\text{-}7}+s_{8\text{-}9} \geq w_{1\text{-}5}$

$s_{1\text{-}3}+s_{1\text{-}4}+s_{0\text{-}1}+s_{0\text{-}3}+s_{2\text{-}3}+s_{2\text{-}6}+s_{6\text{-}8}+s_{8\text{-}7}+s_{7\text{-}9}+s_{5\text{-}9} \geq w_{1\text{-}5}$

$s_{1\text{-}3}+s_{1\text{-}4}+s_{0\text{-}1}+s_{0\text{-}3}+s_{2\text{-}3}+s_{3\text{-}6}+s_{6\text{-}4}+s_{6\text{-}7}+s_{6\text{-}8}+s_{2\text{-}8} \geq w_{1\text{-}5}$

$s_{1\text{-}3}+s_{1\text{-}4}+s_{0\text{-}1}+s_{0\text{-}3}+s_{2\text{-}3}+s_{3\text{-}6}+s_{6\text{-}4}+s_{6\text{-}7}+s_{8\text{-}7}+s_{8\text{-}9} \geq w_{1\text{-}5}$

$s_{1\text{-}3}+s_{1\text{-}4}+s_{0\text{-}1}+s_{0\text{-}3}+s_{2\text{-}3}+s_{3\text{-}6}+s_{6\text{-}4}+s_{6\text{-}7}+s_{8\text{-}7}+s_{7\text{-}9}+s_{5\text{-}9} \geq w_{1\text{-}5}$

Figure 5.7 Demonstration of Using Localities to Restrict the Number of Cutsets Constraints

The second method for limiting the number of cutsets (N_C) is to specify only the "bottle-neck" constraints for each node, as in Figure 5.10. This method is most useful in predominantly end-node limited networks (Chapter 2), where, if enough spare capacity is available near the end-nodes of a span for its restoration, the restoration paths will be able to leave the area of the failure and succeed in restoration because enough spare capacity and nodal degree exist through the rest of the network to connect the path segments near the end-nodes. This method does not truncate the cutsets in any way, so over-constraining does not occur. The parameter specified in this method is the egress path length (EPL), which represents the length of path segment which will allow restoration paths to escape the bottle-neck area. In general, as the average nodal degree of a network increases, the required EPL decreases. An EPL of 1 implies two cutsets corresponding to the incident spans at each end-node.

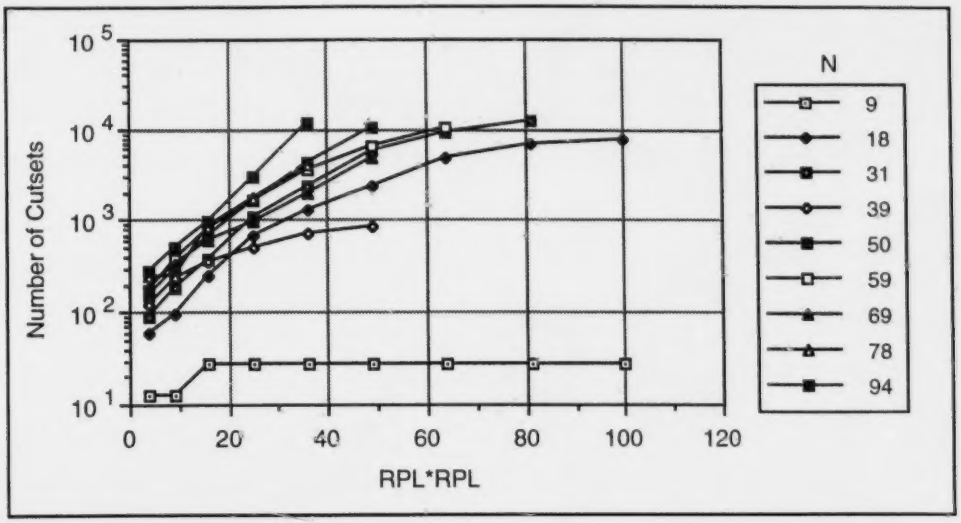


Figure 5.8 Number of Cutsets Generated When RPL Restriction is Used

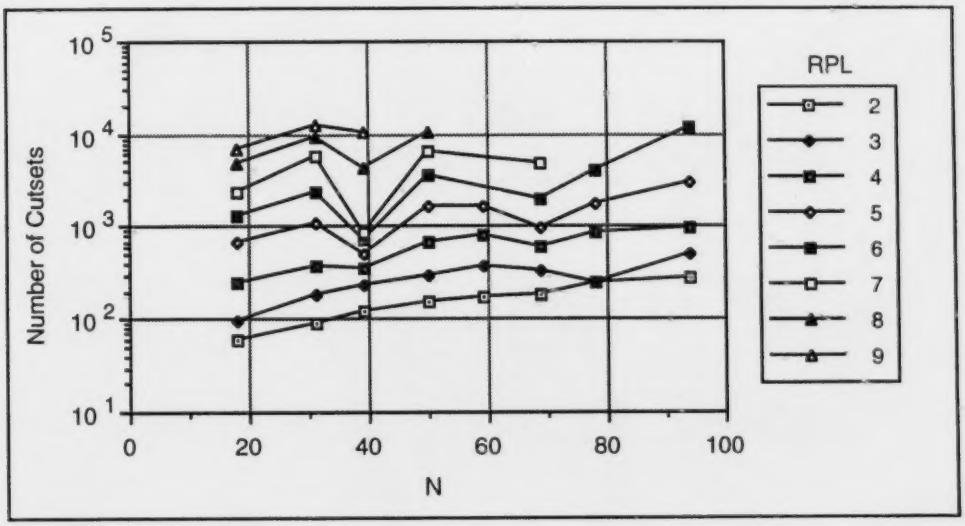


Figure 5.9 Complexity of the Number of Cutsets Generated With Respect to Network Size

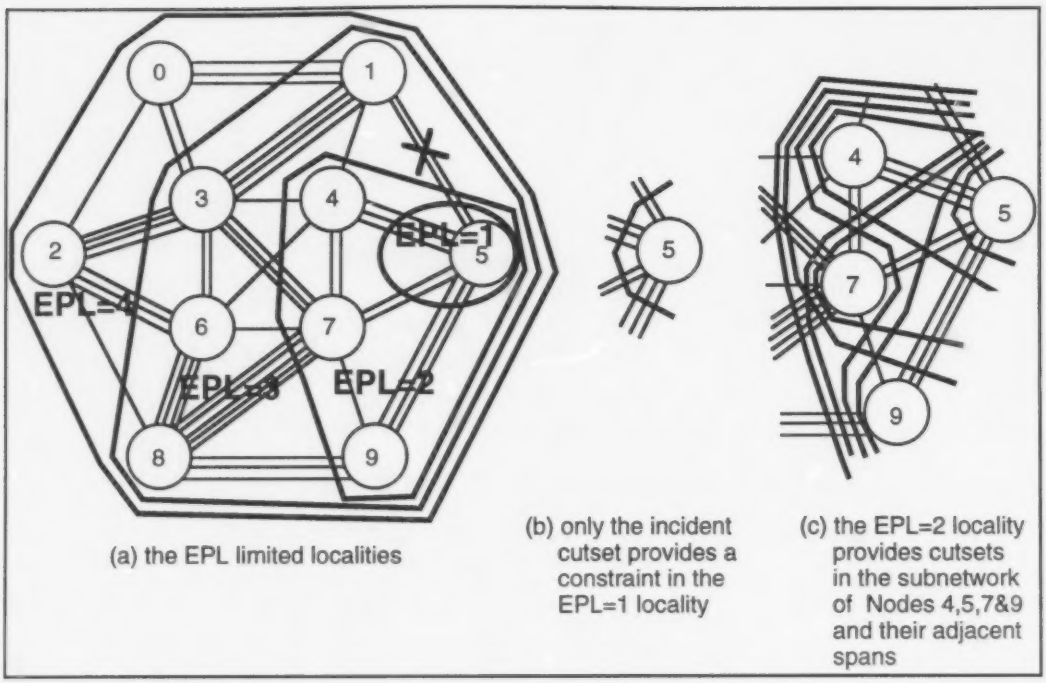


Figure 5.10 Egress Path Length Limited Cutsets

5.3.4 Application of Subsets of Constraints to a Heuristic Implementation

The ICH heuristic (Chapter 6) uses both the EPL and RPL limited cutset principles. The EPL limited constraints do not over-constrain the problem and, therefore, provide a useful initial guess of the constraints which an iterative heuristic requires. Because of the functional limitation of $N_c < 500$, an EPL=1 is probably required that reduces the subset to only the incident cutsets. An iterative heuristic could increase EPL for spans which were not restorable after the initial design stage.

A restoration algorithm may include an inherent restriction on RPL. In these cases, the RPL limited cutsets impose only those desired restrictions on the solution. Thus, the RPL limited cutsets do not impose an over-constraint on the problem, but rather provide a desired result that restoration will be assured within the specified maximum path length objective. In this situation, a heuristic can benefit from RPL limited cutsets. However, $N_{tc} < 500$ restricts the use of the RPL limited cutset principle alone to RPL≤3.

6 Iterative Cutsets Heuristic

The Iterative Cutsets Heuristic (ICH) is the first technique considered for near-optimal solution of the SCP problem. ICH follows the methods reported in [SaNi90]. It applies the cutset flow techniques of the Direct Cutsets Algorithm (DCA), but approximates some DCA functions in order to avoid the high inherent complexity reported in Chapter 4 and adds an explicit RPL limitation feature.

Chapter 5 showed that DCA requires an exponential number of cutsets as constraints to produce a restorable network design. ICH reduces this complexity by using only a strategically selected subset of the cutsets required to fully constrain the IP and produce an optimum solution The next section will describe how this subset of constraints is selected, initially, and after each iteration that does not result in a restorable network design. This section also presents an alternate implementation (ICH RPL) which uses locality information to further restrict computational complexity while limiting observed restoration path lengths.

ICH also reduces the time complexity of DCA by using a Linear Program (LP) to solve for a network design from the constraints, not IP. A proposed network design is obtained from the real valued link quantities produced by the LP, by rounding up the quantities to integral values. This operation compensates for not using an Integer Program (IP) as DCA requires. The IP will produce a minimum design from the constraints, whereas the LP may not, depending upon how many links it assigns non-integer values to. The fractional link difference is small overall in typical networks however. Section 6.5 discusses the choice between IP and LP in more detail.

This chapter discusses the ICH implementation in detail, fully specifying the enhancements over the work presented in [SaNi90]. ICH test results are analyzed in Sections 6.3 through 6.6 on the characteristics of time-complexity, LP versus IP, restoration type and restorability. Presentation of observed execution time of ICH and network growth are deferred until Chapter 8.

6.1 Introduction

ICH's objective function is the sum of all spare capacity. It is minimized subject to a system of constraints that satisfies restoration for working capacity by ensuring adequate minimum cutset flows through the network spare capacity graph. ICH uses an LP to solve the constraint system of (4.3) to (4.5). ICH avoids the exponential number of cutset constraints required to guarantee a fully-restorable network design using a carefully selected subset of cutset constraints. ICH develops this subset of constraints through iteration of the LP, and detection and addition of missing constraints.

A final subset of the full LP constraints achieves a satisfactory network design when the initial set is carefully chosen. The author desires that initial cutsets must: (a) be identified in

polynomial time (with respect to the size of the network), (b) be few in number, and (c) ensure the required minimum restoration capacity for every span. Rules (a) and (b) serve to minimize ICH's complexity. Rule (c) reduces the number of iterations required to find a fully-restorable network design and, hence, also minimize the time complexity of ICH. Rule (c) allows only those constraints that can potentially guide the LP to finding a feasible network design. Thus, in some executions of ICH, it is expected that a fully-restorable network design is achieved in a single iteration. The iteration process is only intended to modify the design in the case where the first iteration does not find a fully-restorable network design.

Based on the criteria just proposed, ICH uses the incident cutsets for every span as the initial constraints. These cutsets can be identified directly from the adjacency list for each span, requiring O(S) time complexity for building the first set of constraints. In a network which is heavily end-node limited, the incident cutsets may provide enough guidance to the LP to lead to a feasible network design in a single step. Therefore, for quasi-planar networks it is desirable to use only the incident cutsets in the first step, because they balance rules (b) and (c) by providing a set of constraints in which the cutsets are uniformly distributed throughout the network and are small in number.

To illustrate, ICH will be used to design a near-optimal SCP in the network of Figure 6.1. The constraints of the LP for the first ICH iteration (the incident cutsets) are included on the figure.

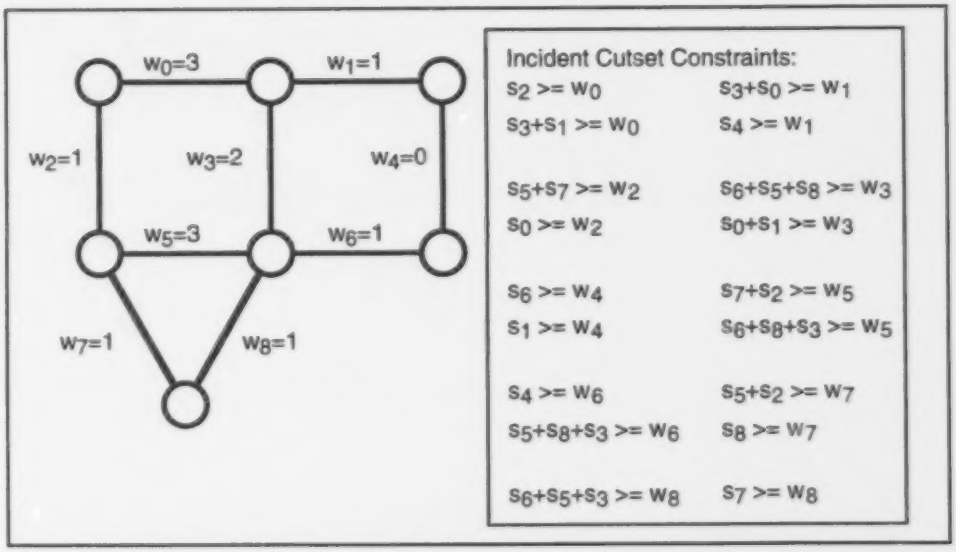


Figure 6.1 The Incident Cutsets Provide the First Set of Constraints

The LP returns a vector of real-valued span capacities which satisfy the constraints. These capacities are rounded up to integer values and, as in [SaNi90], each spare link is tested for its contribution to restorability by removing it and calculating R_n. Links which are not required for maintaining the network restorability are removed before the next ICH iteration. Figure 6.2 displays the spare capacity placement returned in the first iteration of this process. This placement satisfied all of the constraints, but some of the spans are still not restorable. For example, Span 2 has available only two restoration paths, as shown in Figure 6.3(a).

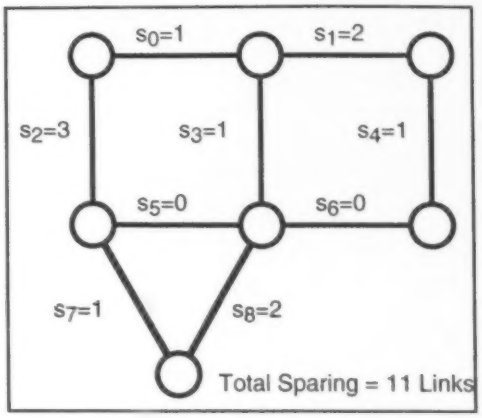


Figure 6.2 The Spare Capacity Placement of First Step LP

When the LP constrained by incident cutsets does not achieve a feasible network design, subsequent steps add constraints for the spans which are not restorable in the first design. Any number of methods could identify additional constraints for subsequent iterations. Figure 6.3 illustrates the method used here, with Span 5 as an example, but constraints are added for spans 0, 1, and 3 as well in this step. ICH identifies additional constraints as follows:

1. Remove the spare links associated with any restoration paths available to the span which is not fully-restorable.
2. Operating on the network of spare capacity, identify the two connected sub-networks, each containing one of the end-nodes of this span.
3. Operating on the full network again, extend the sub-networks identified in (2) to include all network nodes while maintaining two distinct sub-networks.
4. The spans which straddle the border between the two sub-networks identified in (3) represent spans in a new cutset. Add this cutset to the LP constraint set.

Operation one can be performed with metaDijkstra (suggesting k-shortest paths restoration characteristics, see Section 4.1.6.2) or with a maximum flow algorithm [Gibb85] (suggesting max-flow restoration). The two methods had no discernible difference in performance; therefore, ICH uses metaDijkstra due to a lower time complexity of $O(W \cdot N \cdot \log(N))$ as compared to $O(N^3)$ for max-flow. The ICH RPL implementation (Section 6.2) also only uses metaDijkstra for operation one, because it seeks an SCP which is compatible with k-shortest path restoration.

For operations two and three, identification of a connected subnetwork can be obtained with either DFS (see Section 4.1.6.1) or Dijkstra. ICH uses DFS because of its lower time complexity.

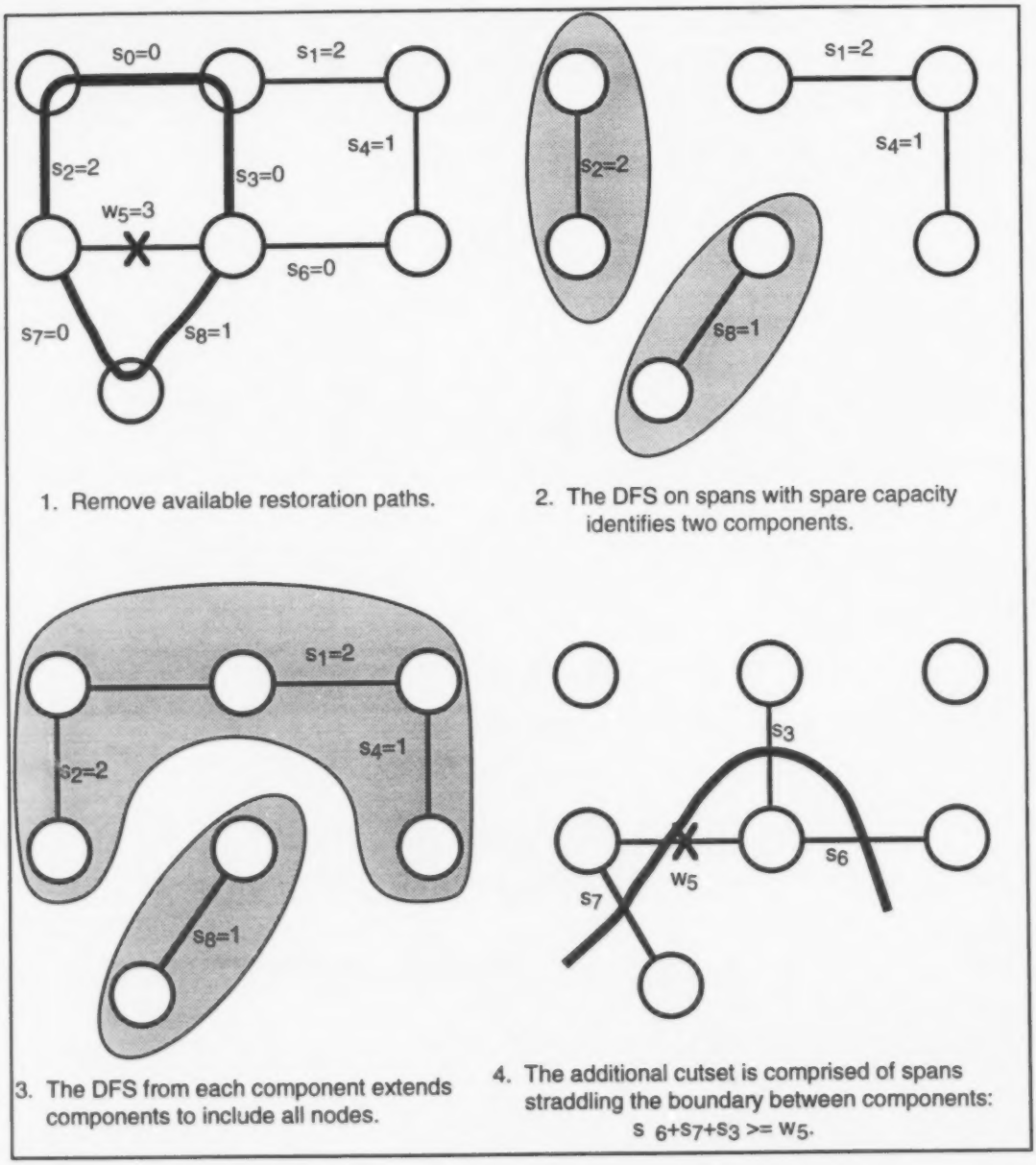


Figure 6.3 Demonstration of Adding a Cutset Constraint for Unrestorable Span 5

In operation three, the ICH algorithm seeks to extend one of the sub-networks to include all of the remaining nodes; only when that fails is the other sub-network extended. This

specification restricts the non-extended sub-network to a smaller geographical area in an attempt to minimize the number of spans of the new cutset without an increase in time complexity of this operation.

At each step, ICH uses the above four operations to add a single additional cutset for each span which is not fully restorable in the network designed by the previous step. Before the second LP execution on the example network, ICH therefore added the following four constraints:

$s_6+s_7+s_3 \geq w_5$ (as in Figure 6.3, operation 4), $s_6+s_0 \geq w_3$, $s_6+s_3 \geq w_0$, $s_6 \geq w_1$

ICH executes the LP again, with these additional constraints and the incident cutset constraints, resulting in the SCP shown in Figure 6.4. This SCP has the same total capacity as the previous design, but the LP has redistributed the spare capacity to conform to the added constraints. But, spans 0 and 7 still are not fully restorable. So, ICH adds two extra constraints and executes the LP again.

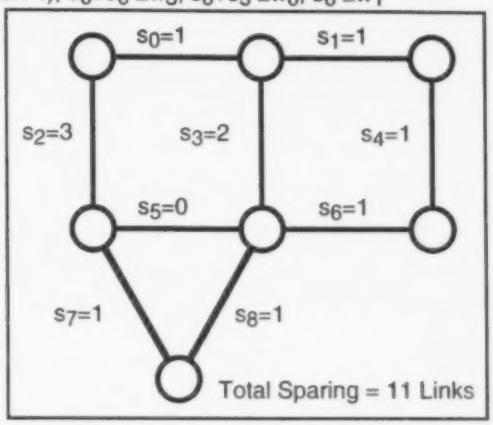


Figure 6.4 The Spare Capacity Placement of Second Step LP

Eventually, after a few steps, the network is fully restorable and the network has a minimum number of spare links. Section 6.4.2 provides data on the number of steps observed before the network design converges to full restorability. For the example network, additional constraints are added in four steps and the final network design has a total of three more spare links than the first LP's SCP, as shown in Figure 6.5.

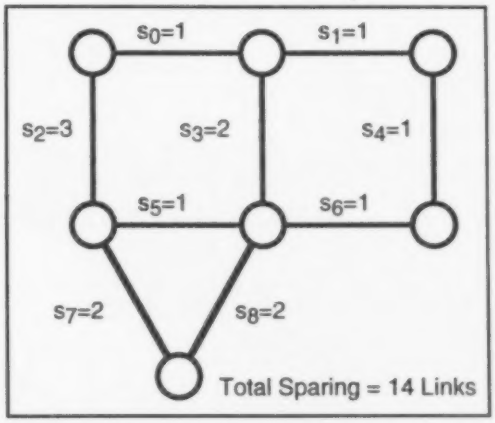


Figure 6.5 The Final Spare Capacity Placement

6.2 ICH With Restoration Path Limit (RPL)

An alternate implementation of ICH, called ICH RPL, includes a partial restriction (not in [SaNi90]) on the length of the restoration paths which are allowed in the network design.

Although ICH guarantees that the required number of restoration paths exist, the individual paths which achieve this flow are not limited in length. Therefore, the restoration process may have to accept a restoration path of any length; otherwise, the network may not be

restorable. This poses a problem when networks must have limited restoration path lengths, such as those which do not have echo control equipment at all locations in the network. ICH RPL attempts to address these concerns.

ICH RPL uses locality information to restrict network operations (see Section 4.1.4 and Section 5.3.3). Specifically, the cutsets ICH uses as constraints for the LP are truncated to include only local spans to the span being restored. Therefore, the spare capacity placement of the LP provides the required flow through the locality, rather than through the entire network as in ICH. However, the RPL limit used to define the locality does not directly restrict the observed restoration path lengths within spans of the locality. This is because the ICH RPL capacity placement is still fundamentally based only on flow-quantity and does not have any inherent restriction on restoration path length, as a k-shortest paths capacity placement would. This is a fundamental limitation on the ICH (and ICH RPL) method. However, ICH RPL achieves some improvements over ICH in controlling the maximal restoration path required for full restorability (see Section 6.6).

Allowing only localities containing the RPL+1 closest nodes to the failed span would strictly limit the restoration path lengths, because any possible path through this area is a maximum of RPL spans in length. But, the lack of route diversity that this produces over-constrains the problem to the point that most of the restoration paths have lengths much smaller than RPL. Such over-constraining ultimately results in a design that is over-provisioned in spare capacity.

As would be expected, ICH RPL network designs have somewhat more spare capacity than the non-limited ICH; the degree of extra spare capacity depends on the RPL constraint. For example, when ICH RPL was executed with RPL=10 for a subset of the study networks (d_{avg} = 3 and 4, N <= 60), it placed1328 spare links (compared with 1275 spare links in the basic ICH implementation), representing an increase of 4.2%.

Both ICH and ICH RPL will be investigated further in comparisons to follow. However, all future references to ICH will concern the basic (non-path length limited) ICH implementation unless specifically stated to be iCH RPL.

6.3 ICH With Transmission System Modularity

The ICH method can accommodate system modularities during the design phase, rather than post processing the design to add system modularities as in [SaNi90]. The algorithm, as described above, is altered so that the LP constraints specify systems (modular bundles of links) rather than individual link quantities. If the network contains various system module sizes, they can be incorporated via multiplicative constants. For example, Figure 6.6 depicts the same network as in figures 6.1 to 6.5, and the corresponding modularity-based constraint system. The

form of the constraints that incorporate modularity appears to be more complex than those of Figure 6.6, but only the s_i are variables, as in the previous format. Therefore, the only changes to the constraints are the multiplicative constants for the s_i elements.

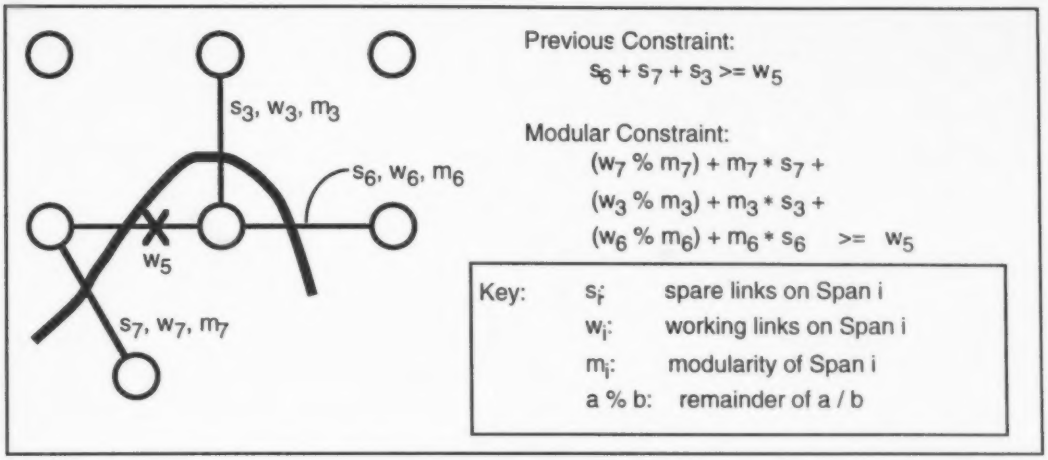

s_3, w_3, m_3
s_6, w_6, m_6
w_5
s_7, w_7, m_7
Previous Constraint:
$s_6 + s_7 + s_3 >= w_5$
Modular Constraint:
$(w_7 \% m_7) + m_7 * s_7 +$
$(w_3 \% m_3) + m_3 * s_3 +$
$(w_6 \% m_6) + m_6 * s_6 >= w_5$
Key: s_i: spare links on Span i
w_i: working links on Span i
m_i: modularity of Span i
a % b: remainder of a / b

Figure 6.6 Incorporation of Modular Systems Into ICH Constraints

6.4 The Theoretical Complexity of ICH

6.4.1 Worst-Case Complexity

There are three primary factors in the execution time of ICH: (1) generation of additional constraints at each step including testing for $R_{s,i} \neq 1$; (2) LP execution time; and, (3) iteration of (1) and (2).

ICH identifies the initial set of cutset constraints (the incident cutsets) from the adjacency lists of the end-nodes of each span. This requires O(S) time. For subsequent steps, ICH performs three computations for each span: (1) k-shortest paths identification of restoration paths to test restorability; (2) DFS identification of subnetworks of spare capacity from end-nodes; and, (3) DFS extension of the 2 subnetworks from (2) to include all nodes.

Using the metaDijkstra shortest paths algorithm to test for $R_{s,i}=1$, the overall complexity of stage one is O(W·N·log(N)), where W is the total number of working links in the network. The DFS has a complexity of O(max(N,S)) [Gibb85] - less than metaDijkstra. Thus, the time complexity of the overall process of identifying new cutsets at each step of the algorithm is O(W·N·log(N)), and is dominated by finding those spans that are only partially restorable.

The most important contributor to ICH time complexity is the execution time of the LP . This is dependent upon the number and size of the constraints. The LP constraint size is measured as the total number of span terms in all cutsets used as constraints. The number of LP

constraints of the first step of ICH is O(S); each constraint contains no more than (d_{max}-1) spans. At each subsequent step, ICH will add no more than S additional constraints (representing the case where no span was restorable). The maximum number of spans in a cutset cannot exceed S. Therefore, the total number of constraint elements in the i-th step of ICH is O(i·S) and the LP constraint size is O(i·S^2).

Here, the ICH heuristic uses a Simplex LP which has a good average execution time, but a worst-case execution time which is exponential in the size of the constraint set. Khachiyan [Khac79] and Karmarkar [Karm84] (summarized in [Schr86]) have developed polynomial-time implementations for LP, but these implementations are not yet competitive with Simplex because of very high execution times (large constant multipliers) [Schr86].

The LP will return a fully-restorable network design if the constraint set includes all possible cutsets. However, it was shown in Section 5.3 that the number of cutsets in a network is exponential in the number of nodes. An LP operating on any subset of this full constraint set may result in a network design which is not fully-restorable. Therefore, ICH could, strictly, require O(2^N) iterations. This establishes the worst-case complexity of ICH as exponential.

6.4.2 Average-Case Complexity

Figure 6.7 displays LP execution time vs. LP constraint size for all fully restorable network designs produced by ICH over the 36 trials networks of Section 3.7.3. The square of the correlation coefficient between the quadratic curve fit and the data is 0.94. A cubic curve fit only results in improvement in the correlation coefficient at the third decimal point. Therefore, the average-case LP execution time is E(n^2), where n is the LP constraint size. However, an important qualification of this result is that only data from successful network designs are present in the analysis. Two examples of unsuccessful LP executions were observed in our experiments:

N=50, d_{avg}=4, 1151 elements, $T_{execution}$ > 500 000 s (5.8 days)

N=30, d_{avg}=6, 2061 elements, $T_{execution}$ > 750 000 s (8.7 days)

In these cases, the LP had to be aborted before obtaining a solution.

The LP constraint size of the first iteration of ICH is E(S·d_{avg}), representing the incident cutsets. Of all ICH network designs, the initial LP constraint size ranges from 52% to 94% of the constraint size of the (final) iteration that resulted in a fully restorable design, as seen in Table 6.1. Thus, the initial constraints provide more than half of the LP constraint requirement in all instances. This average-case analysis shows an increase in constraint size from first iteration to last of E(1) and, therefore, a final LP constraint size of E(S·d_{avg}).

Table 6.1 Final Constraint Size as Compared to Initial Constraint Size

d_{avg}	Initial / Final Constraint Size Ratio				Number of Networks
	Mean	Min	Max	Variance	
3	0.698	0.556	0.782	0.0064	9
4	0.704	0.521	0.862	0.016	8
5	0.828	0.660	0.935	0.015	5
6	0.804	0.768	0.873	0.0022	4

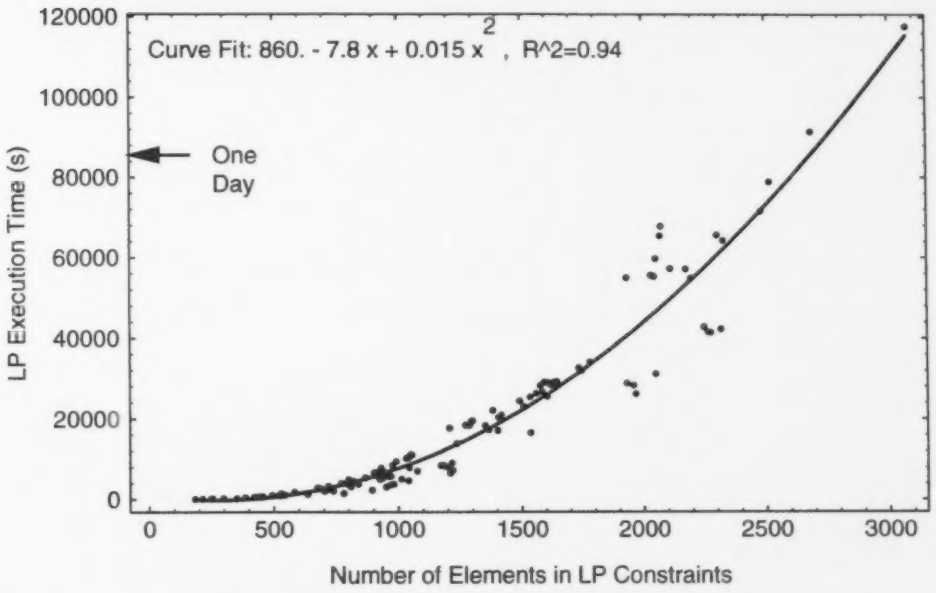


Figure 6.7 Time Complexity of the LP

Table 6.2 shows the number of iterations required by ICH to design a fully-restorable network. The number of iterations decreases as the connectivity of the network increases, reflecting the greater relative tendency for restoration to be end-node capacity limited in high connectivity networks (see Section 1.1), therefore well characterized by the initial set of constraints (the incident cutsets).

Table 6.2 Number of Iterations to Achieve Restorable Network Designs

davg	Number of Iterations				Number of Networks
	Mean	Min	Max	Variance	
3	6.0	5	9	2	9
4	5.25	3	7	1.64	8
5	3.4	2	5	1.3	5
6	3.5	2	5	1.67	4

From Table 6.2, the number of iterations required for ICH to achieve fully restorable designs is E(1). The number of iterations is expected to be constant against network size, because ,with the use of neighborhoods, the number of cutsets is $O(2^{N'}) = O(2^{d \cdot RPL^2})$. In the scenarios investigated here, both RPL and d are constants and therefore E(1) is expected. The actual number of iterations was far below the theoretical bound because restorability within the networks studied is typically limited by end-node bottle-necks. Therefore, the incident cutsets provided as initial constraints are over half of the constraints required in these networks. The process of evaluating restorability and locating additional constraints at each iteration had a negligible contribution to the observed execution times.

Therefore, the overall complexity of an average-case execution of ICH is $E((S \cdot d_{avg})^2)$. The worst-case execution time of ICH is exponential.

These observed execution times for all fully restorable network designs are plotted against the size of the network (number of nodes) in Figure 6.8. For the constant d_{avg} curves, E(S) = E(N) because $S = d_{avg} \cdot N/2$. Therefore, the general result of quadratic complexity with constant node degree can be observed in this figure. As already mentioned, ICH did not complete some of the designs, such as N=50, d_{avg}=4. But, the designs which ICH did complete display the expected quadratic execution time. Figure 6.9 provides better proof of this by dividing the execution times for the d_{avg}=4 networks by the number of steps of ICH required in each case. This does not account for the extra constraints which result from the extra steps; even so, a quadratic curve fit is highly correlated (R^2=0.89) to the data. The constant multiplier to accompany the asymptotic notation can be determined from the equation for the networks in which the quadratic term dominates (N>50) and ,therefore, the execution time is $4.04 \cdot N^2 \cdot$(the number of iterations).

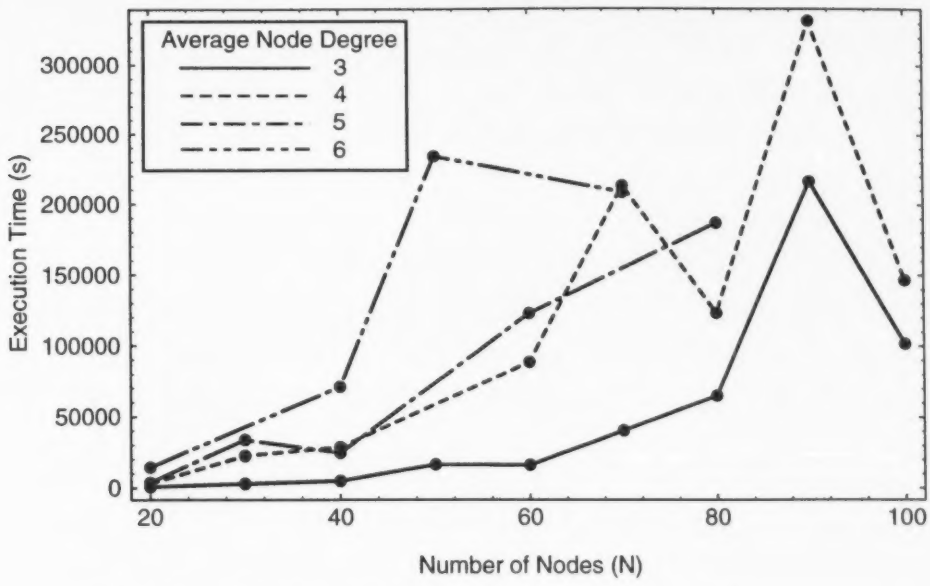


Figure 6.8 Experimentally Observed Time Complexity of ICH

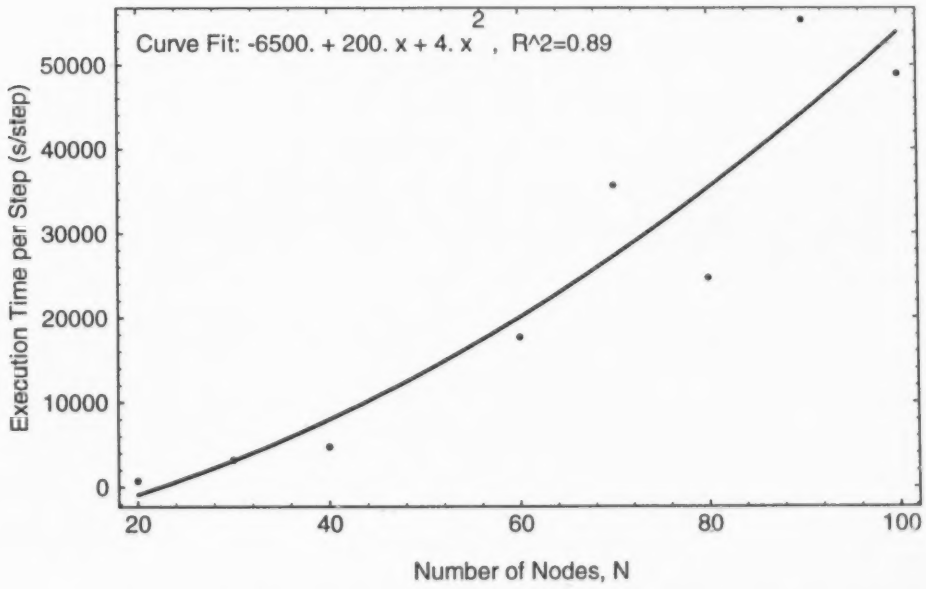


Figure 6.9 Experimental Time Complexity of ICH -- Adjusted for Number of Steps (d_{avg}=4)

6.5 Linear Programming Versus Integer Programming

Strictly speaking, ICH only achieves an optimal network design if it solves the constraints with an Integer Program (IP). However, because Linear Programming (LP) has a lower average-case time complexity than IP (polynomial versus exponential), it is better able to provide the

execution speed which SCP requires. This work uses a readily available Simplex LP implementation which is a module of the Mathematica™ software package. The LP time complexity is, strictly, exponential, but LP implementations are known which execute in polynomial time [Khac79, Karm84, Schr86]. The proof of polynomially implementable LP's is consistent with the observation of low time complexity for an average execution of the Simplex LP. As for IP's, [Schr86] shows that IP's are NP-complete; therefore, no reasonable execution time algorithm is available.

So, how much capacity is added by using LP instead of IP? This question is addressed through experimentation.

An IP was not available for experimentation, but a lower bound on the capacity which either an IP or an LP can place is defined by the real valued solutions provided by the LP. For the 26 study networks in which ICH obtained fully restorable designs, ICH designs required an average of 4.7% more spare capacity than this lower bound, with individual designs ranging from 0% to 13.3% more spare capacity than their lower bounds.

Figure 6.10 compares the spare capacity required by ICH to the real-valued lower bound spare capacity for the degree 4 group of study networks. For these networks, the ICH design averaged 4.4% more capacity than the lower bound, with a variance of 0.04%, and a range from 1.6% to 6.6%.

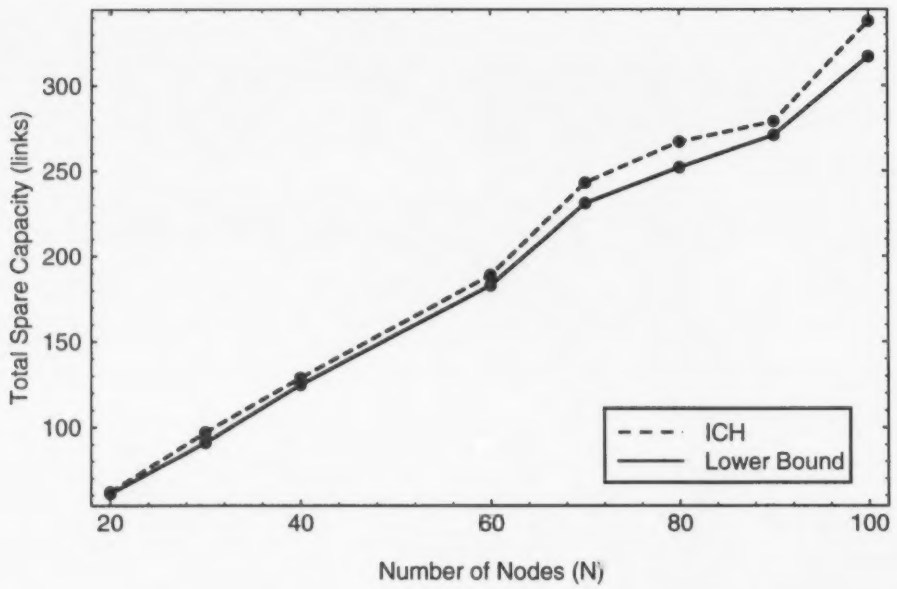


Figure 6.10 Spare Capacity Requirements for Degree 4 Networks

6.6 Restoration Type and Restorability

As with any cutset-based method, ICH seeks to design a network that will provide a max-flow restoration capacity which is at least as large as the potential loss in working capacity for each span. Therefore, ICH will provide a restorable network design only if the restoration algorithm can find paths compatible with max-flow. However, known real-time restoration algorithms more closely mimic a k-shortest paths solution [Grov89], as opposed to max-flow.

ICH incorporates both a k-shortest paths routine for confirming and guiding restorability, and an LP operating on flow constraints. This apparent conflict was introduced to increase the probability that the final network design would be compatible with k-shortest paths restoration. But as a result, the k-shortest paths phase, which tests for restorability between steps, will occasionally reject a network design for which the required max-flow capacity already exists. This rejection will cause an additional cutset (a redundant constraint) to be added to the constraint set which was already satisfied by the previous design. Small changes in the LP's constraints have been observed to drastically change the network design. Generally, a small change to a network design which accommodates max-flow will allow accommodation of k-shortest paths and, therefore, the next step of ICH may return with a network design which does accommodate k-shortest paths flow.

The k-shortest paths phase poses no danger to the reliability of ICH. Whenever two steps of ICH produce the same network design, a known conflict exists between k-shortest paths flow and max-flow; ICH will end with a network design that cannot be fully restored by a k-shortest paths restoration algorithm. Such occurrences are relatively rare, because k-shortest paths and max-flow are equivalent to within a few percent over a wide range of quasi-planar networks [DuGr91].

However, when operating with an RPL, a method has not yet been found to strictly constrain a max-flow algorithm to only consider these length restricted paths. In the ICH RPL implementation, localities provide some restriction to the max-flow path lengths by constraining the paths to routes within the locality. This is an important consideration here because the ICH RPL method is striving to provide restoration path length limited flow with an LP which provides max-flow, only limited in RPL by locality information. Therefore, the difference between max-flow with RPL and k-shortest paths with RPL is expected to be larger than the non-RPL cases considered in [DuGr91] due to an inability to directly impose an RPL limit on the max-flow calculations. Experimental investigation into the restoration path lengths required for restoration in the ICH RPL designs follows.

Tests were performed in which the metaDijkstra k-shortest paths algorithm (see Section 4.1.6.2) attempted to restore the networks designed by ICH RPL. metaDijkstra is constrained to an RPL which is specified by the network designer. For restoration, RPL is usually in the range of

6 to 10. Results show that the networks designed by ICH RPL for an RPL of 10 spans are not fully restorable by metaDijkstra with an RPL of 10 spans. Network restorability statistics for this case appear in the first row of Table 6.3. When metaDijkstra used RPL=10, the networks were an average of 94.5% restorable, with a variance of 0.36%. Indeed, the difference between max-flow and k-shortest paths flow with RPL=10 restriction is 5.5%. Conversely, if ICH network designs are restored by metaDijkstra (which still requires path lengths limited to 10), the resultant restorability statistics are as in the fourth row of Table 6.3. Thus, using ICH RPL instead of ICH improved restorability by k-shortest paths algorithms with RPL=10 from 91.7% to 94.5%.

The RPL limit can be more accurately achieved by designing the network with a tighter specification on RPL than that which will be used by metaDijkstra to restore the network. The effects of this method can be seen by re-examining the restorability of the networks designed by ICH RPL (with RPL=10) when metaDijkstra is allowed RPL limits greater than 10. Statistics for RPL values for metaDijkstra of 11 and 13 appear in Table 6.3 (rows 2 and 3 respectively). When metaDijkstra is allowed an RPL of 13 and ICH RPL uses an RPL of 10, the networks are an average of 99.2% restorable, with a variance of 0.019%. For comparison, the restorability of ICH (no RPL limit) designs by metaDijkstra with these increased RPL limits also appear in Table 6.3: Even with an RPL of 13, metaDijkstra can only provide 97.6% restorability in the ICH designed network.

Table 6.3 Restorability of ICH Designed Networks by a k-shortest Paths Algorithm

design type	RPL	Mean	Variance	Min	Max	Median
ICH RPL	10	94.5%	0.36%	82%	100%	97.0%
	11	96.7	0.14	89	100	98.2
	13	99.2	0.019	95.5	100	99.7
ICH	10	91.7	0.92	76.6	100	94.2
	11	93.9	0.66	80.1	100	98.4
	13	97.6	0.11	92.5	100	100

In conclusion, both ICH and ICH RPL provide max-flow type restoration paths and this does not guarantee full restorability by a k-shortest paths restoration algorithm. Moreover, ICH cannot provide restoration paths which are limited in length, although ICH RPL can approximate this function.

6.7 ICH Summary

Two versions of ICH, (ICH and ICH RPL), were implemented and investigated. Full restoration of the ICH network designs was not possible by a k-shortest paths restoration algorithm, because cutset-based heuristics fundamentally provide max-flow restorability.

ICH RPL uses localities to limit the area of the network through which the restoration flow is guaranteed and as a consequence provides higher restorability in networks where an RPL limit is specified.

The average time-complexity of ICH is polynomial, but the worst-case time complexity is exponential because of the number of steps required to converge, and the use of a Simplex LP.

ICH can accommodate system modularity through multiplicative constants in the LP constraints.

Chapter 8 investigates ICH further, comparing it to SLPA. Chapter 8 investigates the additional topics of network growth, over-restorability of designs and ease of implementation.

7 The Spare Link Placement Algorithm

The Spare Link Placement Algorithm (SLPA) [GrBi90,GrBi91] is the second method considered for near-optimal SCP. SLPA is a heuristic which uses a synthesis-based approach to designing the network. Rather than calculating a network design from constraints, as in the cutset-based methods, the synthesis-based heuristics build a feasible solution to SCP through successive design improvements. The starting point for the design, which is subsequently improved, is specifically chosen for ease of generation and compatibility with the heuristic's approach to network design.

The synthesis-based heuristics guarantee to generate a feasible design, for any network topology, by making restorability (not capacity) the objective function and capacity the constraint (not restorability). For example, in the SCP problem the primary characteristic of a feasible network design is restorability. Like ICH, heuristics of the synthesis type do not guarantee a strictly optimal solution in terms of spare capacity minimization, restorability requirements are indeed achieved.

An alternate synthesis-type heuristic based on Simulated Annealing has been identified and is proposed for future investigation.

Following an introduction to SLPA in Section 7.1, Section 7.2 addresses the important implementation issues and alternatives. Section 7.3 presents a complexity analysis of SLPA with respect to storage requirements and execution time.

7.1 Introduction to SLPA

A network design by SLPA starts with minimal spare capacity corresponding to a restorability far below the restorability objective. SLPA makes successive improvements to the initial design by operating on the spare capacity through the addition, subtraction, or redistribution of spare links. Thus, during the process of synthesizing a network design, SLPA produces a series of intermediate network states which simultaneously approach (a) the restorability objective, and (b) the redundant capacity required in an optimum SCP. The objectives and operations of SLPA can be visualized on the Restorability-Redundancy plane of Figure 7.1. The initial network has both low restorability and low redundancy. It is not known *a priori* how much capacity the final network design requires, but it must meet the restorability objective.

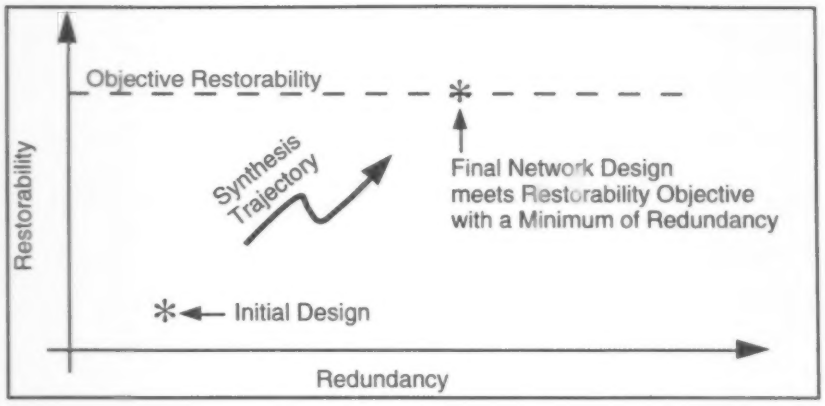


Figure 7.1 The Restorability - Redundancy Plane

Each spare link addition performed by SLPA seeks to maximize restorability with a minimum of additional redundant capacity. Thus, from the initial design point on the restorability-redundancy plane, SLPA seeks a maximum slope ascent in restorability versus redundancy in moving to the next intermediate design state.

SLPA is comprised of two phases executed in series: Forward Synthesis (FS) and Design Tightening (DT).

The first phase of SLPA, FS, performs only operations which add spare capacity to increase restorability. The objective of FS is to attain the network's target restorability level with a near-minimum of spare capacity. Because each operation increases network restorability, FS achieves the target restorability level within W steps, where W is the number of working links in the network.

The second phase of SLPA, DT, only reduces spare capacity. DT uses the operations of redistribution and subtraction of spare links to attain its objective of minimizing spare capacity while maintaining the network restorability at the target level.

Each of FS and DT is a "greedy" algorithm in its own right because (a) each step of a phase performs the same operation on the spare capacity (eg. adding links in FS or deleting links in DT), (b) at each step, a span for the operation is selected based upon a maximum benefit to the objective (a local optimum), and (c) the action performed at each step is not reversed at any subsequent step in the phase.

FS and DT are discussed in more detail in the following subsections. An analysis of the computational complexity of the phases is deferred until Section 7.3.

7.1.1 Initialization

The input to SLPA is a network topology complete with working capacity placements (see Section 2.4). SLPA evenly distributes a small amount of spare capacity throughout the network before the FS phase begins (here, one spare link per span).

The initial spare capacity placement must possess very specific characteristics: Some small amount of spare capacity (compared to the final spare capacity requirements) must be evenly distributed throughout the network. Non-zero initial spare capacity bootstraps the synthesis process, enabling an increase in restorability with a single additional spare link. Without some initial spare links, at least two spare links would have to be added in order to form the first restoration path and thereby increase restorability. The initial spare capacity must be small compared to the final capacity requirements in order to avoid adverse influence of the initial capacity on the final network design. Evenly distributed initial spare capacity may avoid a bias during span selection in the early steps of synthesis. Without even distribution of initial spare capacity, FS would tend to add spare capacity to areas of the network where capacity was already present, and neglect the areas with no spare capacity. SLPA meets these criteria by assigning one spare link to each span in the initial SCP.

7.1.2 FS Phase

Figure 7.2 outlines the strategy of the FS phase. FS does not (directly) try to minimize network cost (total spare capacity), while satisfying restorability as a constraint as ICH did. Rather, it maximizes the restorability benefit for each unit cost (spare link) added to the network. The basic idea is to iteratively address the sub-problem: given one new spare link to "spend", where should it be placed to yield the greatest step increase in R_n?

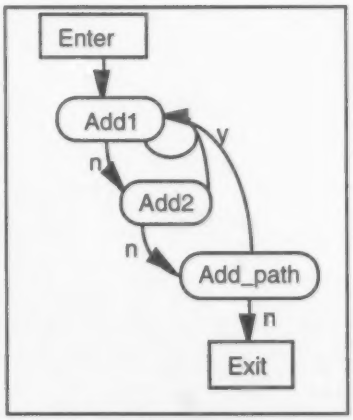


Figure 7.2 Finite State Machine for FS Phase of SLPA

The FS phase proceeds as follows: A spare link is temporarily added to some span, X. R_n is then evaluated. The evaluation uses either a k-shortest link disjoint paths algorithm, or the path-table method that follows, to re-evaluate R_s of all spans individually. The extra spare is moved from span X to another span. R_n is again evaluated. In this way each span of the network is analyzed for the increase in overall network restorability that results from addition of one spare to that span, given the current network state. FS permanently adds a spare link to the span that contributed the greatest increase in R_n. This process makes

intensive use of span restorability calculations. Fast calculation of R_s is therefore given considerable attention in following sections.

FS repeats this single-link investment process, adding one spare link at a time, until there is no span on which an extra link gives an increase in R_n. This can occur for two reasons: (a) Restorability has reached the target level, or, (b) FS has reached a stalling point. If R_n reaches the target restorability level, the FS phase is complete. But if the restorability is below the design target when no one link addition can increase R_n, FS has encountered a "stalling point". This is a situation where FS must simultaneously add more than one spare to increase R_n and get the synthesis going again. In this case, FS analyzes all possible additions of two links for the greatest increase in restorability, and effects the pair of link placements which result in the largest increase in R_n. The algorithm then returns to trying to add one spare at a time.

The recourse to two-link combinations is computationally much more expensive than the basic single-link iteration, but relatively infrequently required. The higher order combinatorial search is appropriate to overcome stalling events, but not for the basic synthesis. Conceptually, FS could include successive recourse to ever higher orders of combinatorial search, whenever the previous level of search stalls. However, this is neither advantageous nor necessary for the problem at hand. After exhausting double link combinations, it is empirically more effective to resort to direct addition of a complete (s,t) path segment to increase restorability rather than search triple, or quadruple, combinations of link additions.

To add a restoration path, FS locates a span that is currently not fully restorable. Then, it explicitly adds one spare link to each span along the shortest restoration route for that span. FS then returns to the one-link-at-a-time mode of synthesis. Links introduced by the process of adding a restoration path are not individually optimal in the sense of greatest global increase in restorability. However, the relative infrequency of this, and the effects of the design tightening phase that follows, makes this unimportant in the final design. Experimental results that follow confirm this.

After the FS phase, the network design has attained the restorability objective; however, as is the case with many greedy algorithms, it may have excess spare capacity compared to the optimum design. At each step in FS, link addition decisions are based on an intermediate network design state, without knowledge of the effect of link additions in later steps. Although all link additions increase restorability by definition, the later link additions could also provide extra protection to an already restorable span. Thus, any step could render a prior addition extraneous, resulting in an excess of spare capacity in the final design. In the restorability-redundancy plane shown in Figure 7.3, the position of the network design after FS is denoted by a star. The figure also shows a typical trajectory through the plane from the initial design to the design after the FS phase.

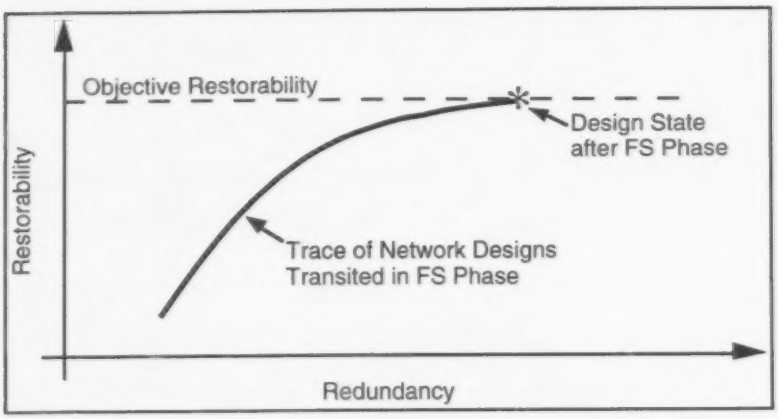


Figure 7.3 The Restorability - Redundancy Plane After the FS Phase

Basic implementations for the FS component modules are shown in figures 7.4 to 7.6. In all of the modules, the subroutine calc_rest() evaluates the proposed operation with a calculation of R_n. In the add_1() and add_2() procedures, bestSpan(s) contains the span(s) which can maximally increase restorability with the addition of a link.

```
add_1() {
    bestRest <- calc_rest();
    foundOne <- FALSE;
    for each (theSpan in S) {
            spare[theSpan] <- spare[theSpan] + 1;
            currentRest <- calc_rest()
            if (currentRest > bestRest) {
                    foundOne <- TRUE;
                    bestRest <- currentRest;
                    bestSpan <- theSpan
            }
            spare[theSpan] <- spare[theSpan] - 1;
    }
    if (foundOne) {
            spare[bestSpan] <- spare[bestSpan] + 1;
    }
    return foundOne;
}
```

Figure 7.4 An Algorithm for Adding One Spare Link

```
add_2() {
    bestRest <- calc_rest();
    foundTwo <- FALSE;
    for each combination (theSpan1 in S AND theSpan2 in S) {
            spare[theSpan1] <- spare[theSpan1] + 1;
            spare[theSpan2] <- spare[theSpan2] + 1;
            currentRest <- calc_rest()
            if (currentRest > bestRest) {
                    foundTwo <- TRUE;
                    bestRest <- currentRest;
                    bestSpans[1] <- theSpan1
                    bestSpans[2] <- theSpan2
            }
            spare[theSpan1] <- spare[theSpan1] - 1;
            spare[theSpan2] <- spare[theSpan2] - 1;
    }
    if (foundTwo) {
            spare[bestSpans[1]] <- spare[bestSpans[1]] + 1;
            spare[bestSpans[2]] <- spare[bestSpans[2]] + 1;
    }
    return foundTwo;
}
```

Figure 7.5 An Algorithm for Simultaneously Adding Two Spare Links

```
add_path() {
    foundPath <- FALSE;
    for each (theSpan in S) {
            spanRest <- metaDijkstra(theSpan)
            if (working[theSpan] > spanRest) {
                    shortestPath <- findPath(theSpan);
                            /* executes dijkstra() and traces pred list */
                    foundPath <- TRUE;
            }
    }
    if (foundPath) {
            for each (theSpan in shortestPath) {
                    spare[theSpan] <- spare[theSpan] + 1;
            }
    }
    return foundPath;
}
```

Figure 7.6 An Algorithm for Adding a Restoration Path

7.1.3 DT Phase

Whereas the FS phase seeks a steepest ascent in restorability against redundancy, the tightening phase changes the direction of travel in the restorability-redundancy state space to whittle away at the redundancy while clamping R_n at the final level achieved by FS.

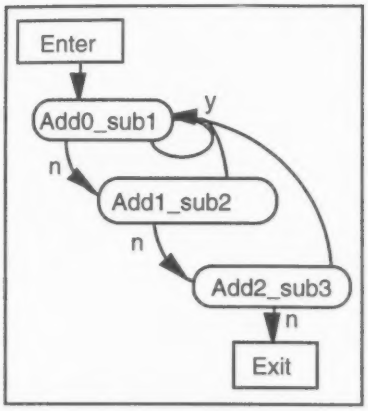


Figure 7.7 Finite State Machine for DT Phase of SLPA

In practice, the FS phase brings the design to a relatively efficient state that meets the target restorability. It serves as a good starting point for a final (limited) combinatorial search for opportunities to "tighten" the solution. Figure 7.7 depicts the DT phase as a Finite state machine. First, add0_sub1(), eliminates any spares which are wholly superfluous, that is, spares which DT can simply remove without any reduction in R_n.

When add0_sub1() can remove no link while maintaining restorability, the tightener examines combinations of capacity-saving redistributions in the spare link assignments. DT uses two levels of combinatorially complete searches for opportunities to add "n" spares while removing "n+1" other spares are used. DT attempts redistribution with n = 1 first (add1_sub2()), and then n = 2 (add2_sub3()). During these searches, DT implements any redistribution which does not reduce R_n before continuing the search. This immediate acceptance of tightening opportunities reduces the number of times that DT executes these searches. Tightening is complete when redistribution at n = 2 cannot remove any further redundancy without decreasing restorability. There is no conceptual reason to stop at n = 2 redistribution, but we found by experimentation that there was little or no improvement after n = 2.

The completion of the DT phase signals the completion of SLPA. Figure 7.8 shows the full synthesis process which results in a near optimal network design.

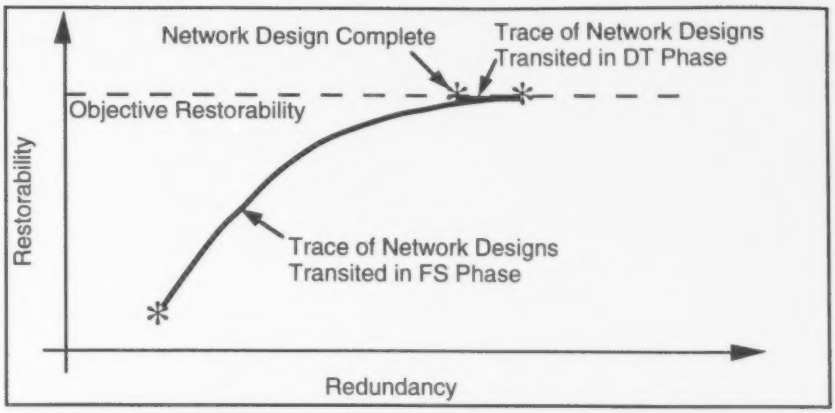


Figure 7.8 The Restorability - Redundancy Plane after the DT Phase

The add2_sub3() state has been observed to find few improvements and, therefore, one of the SLPA implementations to be considered does not include this state. This truncated algorithm is called SLPA Short. The difference in spare capacity required between SLPA and SLPA Short is quantified later in this chapter. These data provide a justification for not implementing more complex (addn_subn+1) states in DT.

Figures 7.9 to 7.11 illustrate basic implementations of the modules used in DT. The algorithms are purely combinatoric in nature, leading to $O(S^5)$ calls to the "calc_rest" procedure for a single execution of the add2_sub3() state. Section 7.3 analyzes optimized versions of these algorithms.

```
add0_sub1() {
        rest <- calc_rest();
        foundOne <- FALSE;
        for each (theSpan in S) {
                spare[theSpan] <- spare[theSpan] - 1;
                currentRest <- calc_rest()
                if (currentRest < rest) {
                        spare[theSpan] <- spare[theSpan] + 1;
                } else {
                        foundOne <- TRUE;
                }
        }
        return foundOne;
}
```

Figure 7.9 An Algorithm for Deleting One Spare Link

```
add1_sub2() {
    rest <- calc_rest();
    foundSet <- FALSE;
    for each (addSpan in S) {
        for each combination (remSpan1 in S AND remSpan2 in S) {
            spare[addSpan] <- spare[addSpan] + 1;
            spare[remSpan1] <- spare[remSpan1] - 1;
            spare[remSpan2] <- spare[remSpan2] - 1;
            currentRest <- calc_rest()
            if (currentRest < rest) {
                spare[addSpan] <- spare[addSpan] - 1;
                spare[remSpan1] <- spare[remSpan1] + 1;
                spare[remSpan2] <- spare[remSpan2] + 1;
            } else {
                foundSet <- TRUE;
        }
    }
    return foundSet;
}
```

Figure 7.10 An Algorithm for Adding One While Removing Two Spare Links

```
add2_sub3() {
    rest <- calc_rest();
    foundSet <- FALSE;
    for each combination (addSpan1 in S AND addSpan2 in S) {
        for each combination (remSpan1 in S AND remSpan2 in S AND
remSpan3 in S) {
            spare[addSpan1] <- spare[addSpan1] + 1;
            spare[addSpan2] <- spare[addSpan2] + 1;
            spare[remSpan1] <- spare[remSpan1] - 1;
            spare[remSpan2] <- spare[remSpan2] - 1;
            spare[remSpan3] <- spare[remSpan3] - 1;
            currentRest <- calc_rest()
            if (currentRest < rest) {
                spare[addSpan1] <- spare[addSpan1] - 1;
                spare[addSpan2] <- spare[addSpan2] - 1;
                spare[remSpan1] <- spare[remSpan1] + 1;
                spare[remSpan2] <- spare[remSpan2] + 1;
                spare[remSpan3] <- spare[remSpan3] + 1;
            } else {
                foundSet <- TRUE;
        }
    }
    return foundSet;
}
```

Figure 7.11 An Algorithm for Adding Two While Removing Three Spare Links

7.1.4 Variants in SLPA Implementation

Experiments show that in a typical 60-node, 120-span study network, 99% of SLPA execution time is spent in the calculation of span restorabilities, $R_{s,i}$. Therefore, performance of SLPA depends significantly on the method used for calculation of $R_{s,i}$. In addition, the last stage of capacity redistribution in the DT phase (add2_sub3()) typically realizes the last few percent increase in capacity efficiency but is relatively costly in execution time. For these reasons, three variants of SLPA are characterized in this work. These are:

SLPA Dijkstra: This is SLPA with the optimized O(N·log(N)) adaptation of Dijkstra's shortest path algorithm to find the k-shortest link-disjoint paths for evaluating R_s. This version also uses capacity redistribution down to add2_sub3() in DT.

SLPA Short: This is identical to SLPA Dijkstra but without use of add2_sub3() module in DT.

SLPA Path-table: This is SLPA using the path-table construct (described in Section 7.2) to efficiently evaluate $R_{s,i}$. This version retains add2_sub3().

7.2 Implementation of SLPA

The SLPA modules presented in Section 7.1 require $O(S^n)$ evaluation of R_n, where n is the number of spans included in an operation (eg. add2_sub3() has n=5). For an estimate of the execution time of SLPA, consider the requirements of the basic implementation of add2_sub3(). Assume that a single machine instruction executes in a micro-second, and metaDijkstra performs the $R_{s,i}$ calculations in the procedure "calc_rest()." The complexity of metaDijkstra is $O(w_{avg} \cdot N \cdot \log(N))$, where w_{avg} is the average number of working links on a span. metaDijkstra must be repeated O(S) times to calculate $R_{s,i}$ for each span. Together with $O(S^5)$ calculations of R_n in add2_sub3(), the total complexity of one call is $O(w_{avg} \cdot N \cdot \log(N) \cdot S^6)$. There are an average of 2.97 working links is per span in the 50 node, 100 span study network. In this form, a single iteration of add2_sub3() would require 26.5 years if only **one** machine instruction performed all of the required work. And this calculation would require 4000 years for the 100-node, 200-span study network.

The SPLA implementation tested here employs numerous techniques that significantly reduce execution times. These include the use of path-tables to reduce redundant operations in searches and span localities to reduce both the combinations of links considered and the length of the searches required.

7.2.1 Span Locality

As was the case with ICH, limiting restoration path lengths (RPL) reduces SLPA complexity. In quasi-planar networks, an RPL specification limits the number of spans that can participate in a given span restoration effort. This locality information (see Section 4.1.4) reduces

the complexity of many aspects of SLPA. For example, calculations of $R_{s,i}$ only include local nodes.

Local spans are an integral part of many of the procedures of SLPA. SLPA initializes data structures containing this information on start-up. The structures locSpan[1..S] and locNode[1..S] correspond to the sets of local spans and nodes for each span, respectively.

When Span A is in Span B's locality, it necessarily implies that the converse is also true: Span B is in Span A's locality. In Figure 7.12, a restoration path for Span A, through Span B, forms a ring including both spans. Therefore, Span B has an equal length restoration path through Span A, as Span A does through Span B. This characteristic makes it equally valid to interpret locSpan[theSpan] in two ways (a) it includes the spans which can assist in restoring theSpan, and (b) it includes all spans to which a change in spare capacity on theSpan would alter their restorability.

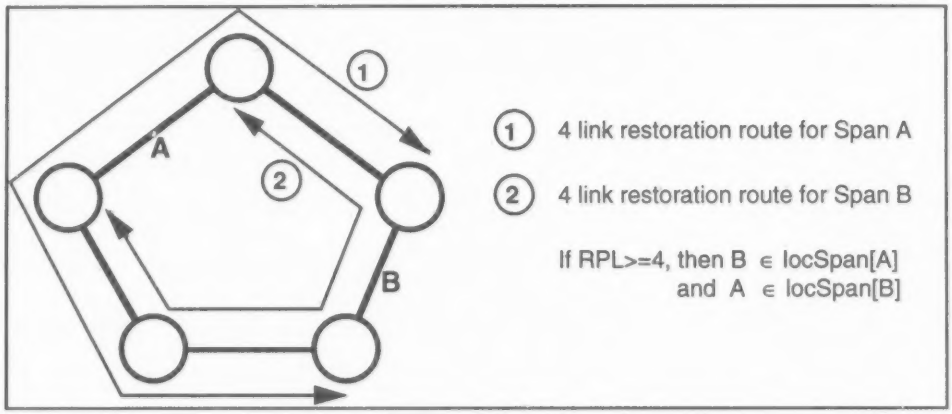


Figure 7.12 Elements of Span Localities

Given the dual use of locSpan, a span-specific RPL scheme [SaVe89] is not possible with the implementation described in this document because the symmetry "A ∈ locSpan[B] if and only if B ∈ locSpan[A]" does not hold. However, an implementation which initializes a different set for each of the two functions of locSpan would not increase the time complexity of the algorithm.

7.2.2 Effect of Span Operations

In order to evaluate the effect of link operations (addition or subtraction) on a single span, SLPA must re-evaluate the restorability of all local spans to it. In more complex procedures like add_2(), SLPA must evaluate the effects of link operations on two different spans simultaneously. In these cases, the union of the two span localities is the operation locality. However, additional information can be used in some cases to reduce an operation locality to less than the union of span localities. For example, when the add_1() procedure is known to have failed before

executing the add_2() procedure, the two added links in add_2() must cooperate if they are to increase restorability. Therefore, in the case of add_2() following an add_1() failure, the locality is the intersection of the individual localities. Applicable localities will be described on a case-by-case basis for the multiple span operations.

7.2.3 Restorability Calculations based on metaDijkstra()

Because metaDijkstra() returns the number of restorable working links for a given span, it can be used as the core of an implementation of calc_rest(), in which here R_n can be assessed as the total number of restorable working links (the sum of the values returned by metaDijkstra() for each span) without normalization to the number of working links in the network. SLPA completes locality information in a preconditioning phase which further reduces the time complexity of the basic implementation. The calc_rest() implementation which uses metaDijkstra for $R_{s,i}$ evaluation and locality information is called "dijk_calc_rest()".

7.2.3.1 Span Localities with metaDijkstra()

For the dijk_calc_rest() implementation of SLPA, the span localities used are approximate sets only. The determination of locality first identifies the local nodes, then uses that information to identify the local spans. A node, say Node 1, is tested for locality to a span, say Span B, by triangulation from the two end-nodes of Span B with shortest path searches. If the searches determine the length of the shortest route from Node 1 to each of the end-nodes of Span B to be x and y spans respectively, then Node 1 is accepted as a local node for Span B if ($x + y < RPL$). This determination may be erroneous if the simple test returns true, but there are no span-disjoint routes for which the criterion is met, that is, the two "exploratory probes" took the same route over some part of their paths. Figure 7.13 illustrates network topologies for which the test would return both correct and incorrect assessments of a local node.

The locality information provided by this method will always include the true local spans and nodes, but occasionally it may also include extra spans and nodes. This approximation does not have an impact on the accuracy of the solution, but tends to increase the computation time. Such inaccuracies only occur where the path diversity is low; therefore, they occur more commonly in the low average node degree networks. Fortunately, execution speed is less of an issue for lower degree networks.

Local span information can be ascertained from the local node information in that any span which is adjacent to two local nodes is considered to be a local span. Therefore, the localities may contain extra spans where extra nodes exist.

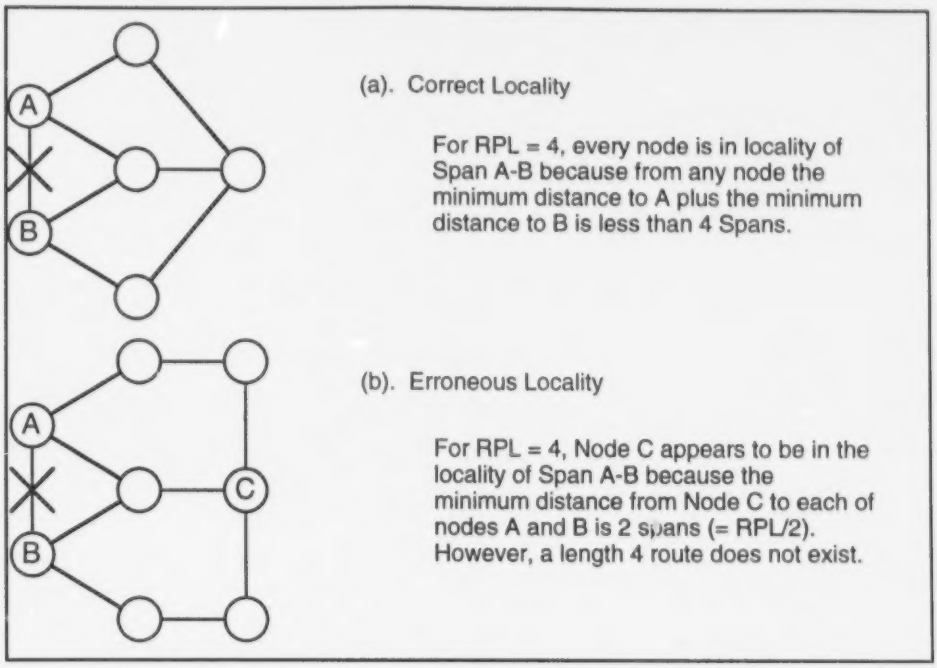

A
B
(a). Correct Locality
For RPL = 4, every node is in locality of Span A-B because from any node the minimum distance to A plus the minimum distance to B is less than 4 Spans.
A
C
B
(b). Erroneous Locality
For RPL = 4, Node C appears to be in the locality of Span A-B because the minimum distance from Node C to each of nodes A and B is 2 spans (= RPL/2). However, a length 4 route does not exist.

Figure 7.13 Identification of Nodes in a Locality

7.2.3.2 The dijk_calc_rest() Procedure

Throughout SLPA execution, accurate values for span restorabilities (spanRest[1..S]) and network restorability (networkRest) are maintained for the current network design. Therefore, for each operation on the spare capacity made by SLPA, network restorability must be assessed by recalculating restorability for spans that are local to the change. The SLPA module passes the list of spans (recalcSpans) which require recalculation of restorability to dijk_calc_rest(). In order to initialize the $R_{s,i}$ for this R_n update, SLPA must execute dijk_calc_rest() once at startup with recalcSpans being the set of all spans.

The dijk_calc_rest() procedure is documented in Figure 7.14. The nodeA[i] and nodeB[i] elements return the end-nodes of span i. The implementation of metaDijkstra() is as presented in Section 4.1.6.2, with the addition of an early termination when the number of restoration paths identified exceeds the number of working circuits to be protected.

The call to metaDijkstra has time complexity $O(w_i \cdot N \cdot \log(N))$ for calculating $R_{s,i}$. Therefore, calculating R_n can require recalculation for every span and has time complexity $O(W \cdot N \cdot \log(N))$ in general.

In a specific calculation of R_n where locality information is used, metaDijkstra only uses local nodes (numbering N') in each $R_{s,i}$ calculation and only calculates $R_{s,i}$ for S_c spans (the

spans which may have been affected by (ie., are in the locality of) the last spare capacity operation). Therefore, the time complexity of dijk_calc_rest() is $O(w_{avg} \cdot N' \cdot \log(N') \cdot S_c)$.

```
dijk_calc_rest(recalcSpans, networkRest, spanRest)
{
    for all (calcSpan in recalcSpans) {
            networkRest <- networkRest - spanRest[calcSpan];
            source <- nodeA[calcSpan];
            target <- nodeB[calcSpan];
            spanRest[calcSpan] <- metaDijkstra(source, target,
work[calcSpan]);
            networkRest <- networkRest + spanRest[calcSpan]
    }
    return networkRest;
}
```

Figure 7.14 The Dijkstra-Based Restorability Calculation Procedure

7.2.4 Restorability Calculations based on a Path-Table

SLPA requires up to one million recalculations of network restorability for complete synthesis of a 60-node 120-span network. Therefore, even though the dijk_calc_rest() procedure has been implemented efficiently, SLPA would benefit from any further decrease in complexity.

When an algorithm executes a single procedure repeatedly, it is often feasible to accelerate the algorithm with preprocessing operations of a higher complexity than a single execution of the procedure itself. The amount of extra complexity which can be incorporated into the preprocessing phase while still achieving overall savings depends upon the number of times the important procedure must be executed. In the case of SLPA, the fundamental component is the recomputation of R_n for every alteration made to the network (calc_rest()). In fact, an execution profile showed that SLPA spends 99.9% of its execution time in calc_rest().

Thus, considering a million calls to calc_rest() comprising 99.9% of SLPA's execution time, preconditioning can significantly benefit SLPA if it can achieve any reduction in the execution time of calc_rest().

The path-table implementation improves the speed of calculating $R_{s,i}$ by use of a pre-processed table of all topologically possible restoration routes. The path-table is storage intensive in that it contains each possible route of all potential span failures. The information is sorted, compressed and saved in a data base. Restriction of the restoration path length (RPL) is a necessary aspect of the path-table method in order to restrict the overall space complexity.

Whether or not a span restoration can use a particular route when the network is in a particular state depends on the current spare capacity dimensioning of the spans of the route. Entries in the path-table are potential routes that must be tested for capacity at look-up time. This

rules out the possibility of O(1) look-ups, because each entry must be evaluated for its minimum route capacity, rather than just indexed. However, the path-table keeps routes sorted by length and statistical frequency of use, and it avoids repeated searches to rediscover routes.

7.2.4.1 The Structure of the Path-Table

Traversing the path-table in calculating $R_{s,i}$ is similar to the operation performed by a Depth First Search (DFS, Section 4.1.6.1). But by using a path-table, many repetitive operations can be eliminated because of the knowledge in the table, and through sorting and merging of the full list of routes.

There is one path table for each span, enabling fast calculation of $R_{s,i}$. Within each table, the first step in creating the path-table is to generate every topologically possible restoration route within the RPL specified. This is an exponential search; therefore, RPL must be limited to a reasonable value, based upon the average node degree of the network. The potential paths are sorted by logical length (ie. number of spans). After this first step, the structure has the form in Figure 7.15. The first row in the structure represents the spans along one of the shortest restoration routes. The last row in the list contains one of the longest restoration routes, which is at most of length RPL. This step of sorting paths by logical length will effect a k-shortest paths flow when calculating $R_{s,i}$ by searching from the top of the table to the bottom. Thus, the path-table combines the speed advantages of a depth first search and the path length characteristics of an k-shortest path search.

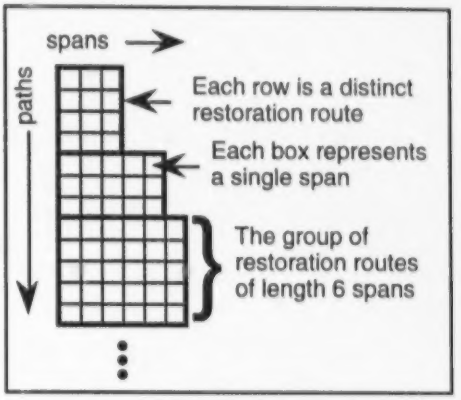

spans
paths
Each row is a distinct restoration route
Each box represents a single span
The group of restoration routes of length 6 spans

Figure 7.15 A List of All Restoration Routes Available for Restoration of a Span After Sorting by Length.

Next, the most common span (ie., the most frequently appearing span amongst the routes of a given length) (bestSpan) is identified, from all the spans on all the routes of length n. The bestSpan achieves its preferred status because it is either a crucial span to the restoration effort, or it is a geographically well-situated span to assist in the restoration effort. Because of its

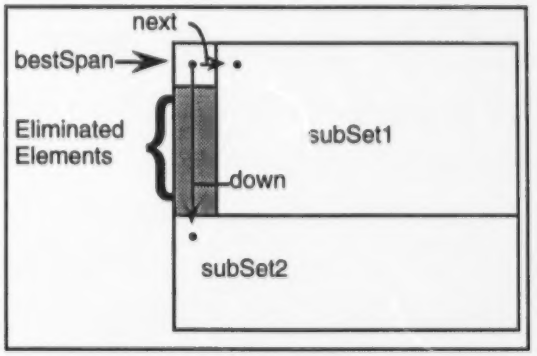


Figure 7.16 Sorting and Reducing Within a Common Route Length

importance, the path-table is arranged so that bestSpan will only be checked once for available spare capacity. For example, if bestSpan had a spare capacity of zero, it would be known with no further tests that all potential paths of that subset yield no restoration paths in the current design state. Therefore, the data structure for length n routes can be divided into three sections: (a) bestSpan, (b) a set of route segments which constitute the remainder of the routes which contain bestSpan (subSet1), and (c) the set of routes which did not contain bestSpan (subSet2). The bestSpan section contains a single element and pointers to subSet1 and subSet2. Figure 7.16 depicts this stage of processing.

Each of the two subsets created in step two (subSet1 and subSet2) can now isolate their most prevalent remaining span. For example, the structure of Figure 7.17 results after performing the first step of recursion (splitting subSet1 and subSet2). This process continues recursively within subsets, until only the individual elements remain. It is these individual elements remaining after the recursive reduction of the data structure which form the path-table.

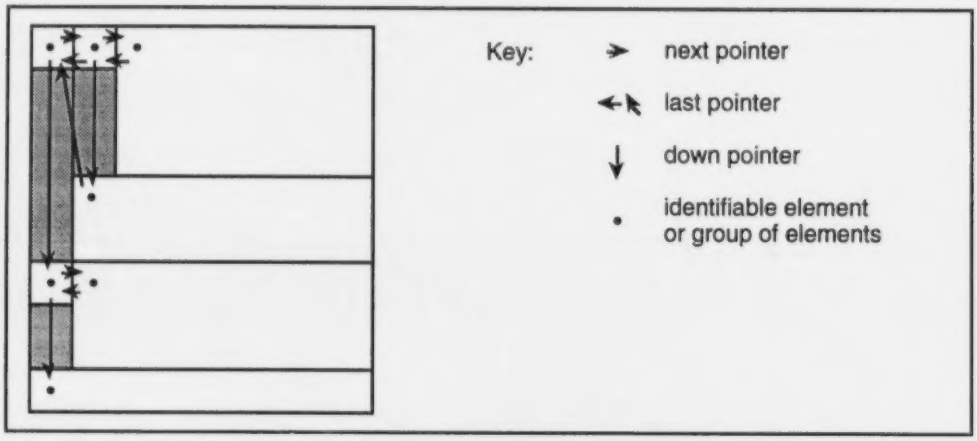


Figure 7.17 Recursive Sorting and Merging Within a Common Route Length

7.2.4.2 Evaluating $R_{s,l}$ With the Path-Table

A procedure called span_rest_pt() performs the same function as metaDijkstra(), but it deduces restorability by traversing the path-table. The path-table traversal within a common path length is a similar operation to a DFS algorithm across the spans of the path-table. It seeks to find a restoration path, by assessing the spare capacity on every span of the path, before considering alternate path segments.

RPL is an inherent part of the path-table; therefore, the locality restrictions are not addressed directly, as they are with the metaDijkstra-based implementation. Locality information

(locSpan and locNode lists) can be incorporated while the path-table is built. Therefore, the locality information will be strictly accurate in this implementation.

The search for restoration paths traverses the path-table from left to right across the "next" pointers. The restoration chain contains the latest span traversed in each column of the table to the left of the current column. The objective of traversal is to find a restoration chain that contains all spans of an entire restoration route and in which each column is represented by a span containing spare capacity; these spans embody an available restoration route. A span with no spare capacity blocks traversal in that direction. For example, in the case of a blocked "next" element, an alternate span for that column is pursued by moving through the "down" pointer. If no "down" pointer exists at the blocked element, the current chain cannot be completed and alternate segments of the chain must be sought.

The search uses "last" pointers when back-tracking to locate an alternate chain from which the traversal can continue. Through back-tracking, previous columns of the chain (to the left) are reassessed for alternate spans with which to continue the current restoration chain. Back-tracking proceeds until a non-NULL "down" pointer is encountered, which indicates that the forward traversal can continue from this element of the chain.

The search identifies a viable restoration route if the chain includes spans from each column. This corresponds to the case in which the search reaches the rightmost column of the table, as indicated by a "next" pointer equal to NULL. The minimum capacity on any span of the restoration chain dictates the number of restoration paths available on the identified route. The traversal algorithm removes the spare capacity consumed by accepting restoration paths before continuing to search for further restoration paths.

The search for the next restoration path does not start from the head of the path-table, nor does it necessarily start from the leftmost column. Instead, the search continues from the right-most column where the previous restoration path was identified. The route acceptance can exhaust one or more of the spans involved in the previous restoration chain and, if so, the traversal algorithm uses back-tracking to find a feasible point to resume forward traversal. This forward traversal continues from an element in the previous chain where all columns to the left of the element contain spans with spare capacity. This form of back-tracking is facilitated through the array "depth[column]," which holds the minimum spare capacity on any span of the chain to the left of the current element. For example, upon removal of n restoration paths from the restoration chain, each column's depth value is decremented by n. The right-most column with a positive depth value anchors the resumed search.

In the span_rest_pt() procedure in Figure 7.18, the variable "ptr" is used to maintain the current position of the search in the path-table. The name of the span contained in a table

element is retrieved through the "span" pointer (ie. ptr->span). The variable "level" corresponds to both the column number and the current number of elements in the chain.

7.2.4.3 Evaluating R_n With the Path-Table

The major advantages of using the path-table rather than metaDijkstra are threefold. First, the search finds all possible paths in a single traversal (iteration). Second, the search examines the most common spans amongst all routes first, regardless of their geographical location with respect to the end-nodes, thereby avoiding consideration of less frequently used spans until the need arises. Third, the search traverses spans that are common to many routes only once.

In other words, spans which appear most often in potential restoration routes are first in the path-table. Thus, when the network is in a highly-restorable state a restorability calculation typically only requires looking at the first few elements in the table. This is an especially important observation and property because much execution time is spent in the terminal regime of approach to minimum capacity at 100% restorability (in DT). When just two or three most-frequently used spans have spare capacity, a plethora of alternative routes which contain this span are enabled and full restorability can usually be verified without further search. This is a useful property, because SLPA moves very quickly to network average restorability in excess of 80%. To calculate $R_{s,i}$ using the path-table, the traversal continues until either establishing $R_{s,i} = w_i$ or exhausting the path-table.

The pt_calc_rest() procedure in Figure 7.19 assesses $R_{s,i}$ by using span_rest_pt() as the primary function call. Note the similarity to dijk_calc_rest().

```
span_rest_pt(calcSpan, work) {
    for all (theSpan in Spans) spare[theSpan] <- conSpare[theSpan];
    ptr <- pathTable[calcSpan];
    level <- 0; spanRest <- 0;

    while ((NOT ptr = NULL) AND (spanRest < work)) {
        if (level = 0 OR (spare[ptr->span] < depth[level-1]))
            depth[level] <- spare[ptr->span];
        else    depth[level] <- depth[level-1];

        if ((NOT ptr->next = NULL) AND (NOT depth[level] = 0)) {
            /* advance to next column */
            ptr <- ptr->next;
            level <- level + 1;
        } else if (NOT depth[level] = 0) {
            if (depth[level] + spanRest > work)
                depth[level] <- work - spanRest;
            spanRest <- spanRest + depth[level];

            acceptPaths(depth[level], ptr, level);
            backTrack(&ptr, level): /* subroutine modifies ptr */
        } else {
            backTrack(&ptr, level): /* subroutine modifies ptr */
        }
    }
    return spanRest;
}
acceptPaths(npath, backptr, i) {
    while (NOT backptr = NULL) {
        depth[i] <- depth[i] - npath;
        spare[backptr->span] <-
            spare[backptr->span] - npath;
        i <- i-1;
        backptr <- backptr->last;
    } /* end while */
} /* end acceptPaths*/
backTrack(ptr, depth) {
    found <- FALSE;
    while (NOT ptr = NULL AND found = FALSE) {
        if ((NOT ptr->down = NULL) AND
            ((level=0) OR NOT (depth[level-1] = 0))) {
            ptr <- ptr->down;
            found <- TRUE;
        } else {
            ptr <- ptr->last;
            depth <- depth - 1;
        }
    }
}
```

Figure 7.18 The Path-Table-Based Span Restorability Calculation Procedure

```
pt_calc_rest(recalcSpans, networkRest, spanRest) {
    for all (calcSpan in recalcSpans) {
            networkRest <- networkRest - spanRest[calcSpan];
            spanRest[calcSpan] <- span_rest_pt(calcSpan,
conWork[calcSpan]);
            networkRest <- networkRest + spanRest[calcSpan]
    }
    return networkRest;
}
```

Figure 7.19 The Path-Table-Based Network Restorability Calculation Procedure

7.2.4.4 Computational Complexity Associated with the Path-Table

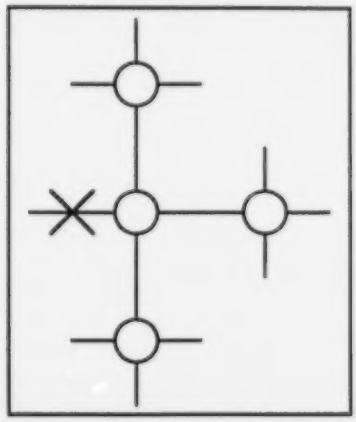


Figure 7.20 Branching of Path Possibilities

The complexity of path-table preparation is linear in the number of spans in the network. For each span, a table is built which specifies all topologically possible restoration routes for that span. To find all possible distinct restoration routes for a given span, an algorithm first selects one of the end-nodes of the span, and searches through the network until encountering the node at the other end of the span. Consider the case where all nodes in the network have the same degree, d. As depicted in Figure 7.20, all (d-1) unfailed spans leaving a node are recorded as beginning possible restoration paths. At each successive step away from the starting node, the number of possibilities increases exponentially by a factor of (d-1). This process is limited by the restoration path limit, RPL, so that the maximum number of steps needed to create the path-table is $(d-1)^{RPL}$. Thus, the worst-case complexity of building the path-table is $O(S \cdot d^{RPL})$, linear in S. The dependence on d and RPL is to some extent under the control of the network designer, as these are engineering parameters. For networks with varying node degrees, the worst-case time complexity of building the path-table is $O(S \cdot d_{max}^{RPL})$, where d_{max} is the maximum node degree in the network.

The path-table also requires $O(S \cdot RPL \cdot d_{max}^{RPL})$ space, where each route requires RPL bytes of storage. Therefore as the average degree of the nodes in the network increases, use of the path-table method may be limited by the available memory.

The worst-case complexity of calculating $R_{s,i}$ from the path-table is a search of the entire table. This is an $O(S \cdot d^{RPL})$ operation for $R_{s,i}$ calculation, so it is $O(S^2 \cdot d^{RPL})$ for R_n calculation.

7.2.5 Implementation of the SLPA Modules Using Locality Information

The specific implementations of the procedures presented in Section 7.1 are presented here. They are analyzed for two components of their time complexity: (a) the extent of search (the number of alternatives considered in a single iteration), and (b) the number of spans requiring calculation of $R_{s,i}$ for each alternative considered (S_c).

7.2.5.1 Add One Spare Link

Figure 7.21 presents the most highly utilized spare link operation within SLPA, "add 1 spare link" (referred to as add_1()).

The (change in) R_n is calculated at line 7, once for each span in the network (S); therefore, the extent of search is O(S). For each proposed span addition, the other spans requiring calculation of $R_{s,i}$ will be spans within the locality of theSpan (locSpan[theSpan]), because an additional link on theSpan can affect the restorability of any span within its locality. Therefore, $S_c = O(S')$.

```
add_1(networkRest, spanRest) {
1       bestRest <- networkRest;
2       foundOne <- FALSE;
3       for each (theSpan in S) {
4               spare[theSpan] <- spare[theSpan] + 1;

5               tempNetworkRest <- networkRest;
6               tempSpanRest <- spanRest;
7               currentRest <- calc_rest(locSpan[theSpan], tempNetworkRest,
                                         tempSpanRest)

8               if (currentRest > bestRest) {
9                       foundOne <- TRUE;
10                      bestRest <- currentRest;
11                      bestSpan <- theSpan
12                      bestSpanRest <- tempSpanRest;
                }
13              spare[theSpan] <- spare[theSpan] - 1;
        }
14      if (foundOne) {
15              spare[bestSpan] <- spare[bestSpan] + 1;
16              spanRest <- bestSpanRest;
17              networkRest <- bestRest;
        }
18      return foundOne;
}
```

Figure 7.21 Add One Spare Link Module of SLPA with Locality Considerations

7.2.5.2 Add Two Spare Links Simultaneously

"Add two spare links", add_2(), is the second most commonly executed module in SLPA, and is similar to the add_1() module analyzed in the previous subsection. Figure 7.22 presents the implementation-specific add_2() procedure.

The complexity of the basic algorithm presented in Section 7.1.1 has been reduced here by restricting the pairs of spans for consideration (the extent of search). Because add_1() has already failed when add_2() is called, only those additional spare links which can cooperate will increase network restorability. Complementary (or cooperating) spans are those that are close enough to each other to form parts of the same restoration path; therefore, complementary spans will be in each other's localities. (theSpan2 is a member of locSpan[theSpan1], and vice-versa.) The add-2() procedure selects theSpan1 from all spans, and theSpan2 from only locSpan[theSpan1]; therefore, the extent of search is $O(S \cdot S')$.

At line 10 (Figure 7.22), R_n is calculated for each of the pairs to be considered. The set of spans for which $R_{s,i}$ requires recomputation (numbering S_c) will be the union of the local spans for each of theSpan1 and theSpan2. The number of elements in locSpan[theSpan1] plus locSpan[theSpan2] is $S_c = O(2 \cdot S') = O(S')$. This specification of S_c is complicated and requires an explanation: The set of spans which theSpan1 and theSpan2 can cooperate to restore is the intersection of locSpan[theSpan1] and locSpan[theSpan2]. However, because path selection order can impact restorability by k-shortest paths, the addition of a single link to theSpan1 can also result in a decrease of restorability to any span in its locality (locSpan[theSpan1]). This restorability decrease (even though spare capacity is increased) is the result of a "trap," whereby restorability is affected by the path selection order (see Figure 3.8) Therefore, restorability recomputation must consider all spans in the localities of either theSpan1 or theSpan2.

A similar analysis can be used for add_n() to show that the extent of search is $O(n \cdot S \cdot S'^{n-1})$ and S_c is $O(n \cdot S')$. Here, the linear part of the dependence on n can be removed by calculating R_n in stages as will be discussed in detail in Section 7.2.5.4.

```
add_2(networkRest, spanRest) {
1       bestRest <- networkRest;
2       foundTwo <- FALSE;
3       for each (theSpan1 in S) {
4               for each (theSpan2 in locSpan[theSpan1]
5                       such that (theSpan1 < theSpan2)) {
6                       spare[theSpan1] <- spare[theSpan1] + 1;
7                       spare[theSpan2] <- spare[theSpan2] + 1;

8                       tempNetworkRest <- networkRest;
9                       tempSpanRest <- spanRest;
10                      currentRest <- calc_rest(locSpan[theSpan1] +
                                locSpan[theSpan2], tempNetworkRest, tempSpanRest)

11                      if (currentRest > bestRest) {
12                              foundTwo <- TRUE;
13                              bestRest <- currentRest;
14                              bestSpan1 <- theSpan1
15                              bestSpan2 <- theSpan2
16                              bestSpanRest <- tempSpanRest;
                        }
17                      spare[theSpan1] <- spare[theSpan1] - 1;
18                      spare[theSpan2] <- spare[theSpan2] - 1;
                }
        }
19      if (foundTwo) {
20              spare[bestSpan1] <- spare[bestSpan1] + 1;
21              spare[bestSpan2] <- spare[bestSpan2] + 1;
22              spanRest <- bestSpanRest;
23              networkRest <- bestRest;
        }
24      return foundTwo;
}
```

Figure 7.22 Add Two Spare Links Module of SLPA With Locality Considerations

7.2.5.3 Delete One Spare Link

The add0_sub1() module has equivalent complexity to add_1() for a single execution. In the add0_sub1() module, a single execution performs all possible single link deletions as they are identified and is, therefore, executed a small number of times. This module will not contribute to the complexity of SLPA because of its relatively low complexity.

7.2.5.4 Add One Spare, While Deleting Two Spares

Figure 7.23 presents the module which "adds one while deleting two spare links". Several aspects of this implementation lead to a reduced complexity from the basic implementation: (a) It assumes that add0_sub1() has already failed to find any single links for removal; (b) It uses locality information wherever possible; And, (c) it calculates R_n in stages. Again, as in add_2(), operation on any span in the network, whether an addition or deletion of

spare capacity, can result in a decrease in restorability due to the dependence of k-shortest paths flow on selection order.

For the following discussion, it is convenient to identify the concept of extended local spans: The union of the localities of the spans in theSpan's locality comprises the extended locality. Spans in the extended locality are called the extended local spans. A span which is in the extended locality of theSpan is contained in the set $loc^2Span[theSpan]$ or, equivalently, locSpan[locSpan[theSpan]].

The number of spans in $loc^2Span[addSpan]$ is larger than the number of spans in locSpan[addSpan], but by how much? In a planar lattice network, if the locality is considered as the spans within a circle of a radius r, then, the extended locality will contain no more than the number of spans within a circle of radius 2·r, because each span along the perimeter of the locality has its own locality of radius r. Based upon the resultant increase in area from a doubling of the radius, the total extended area is less than or equal to four times the area of the single locality. Although this result is only strictly true for planar networks with constant node degree, it is a useful approximation for quasi-planar networks.

Hence: $O(|loc^2Span[addSpan]|) = O(4 \cdot |locSpan[addSpan]|) = O(4 \cdot S') = O(S')$.

The add1_sub2() procedure performs three link operations: (a) Add a link to addSpan, (b) delete a link from subSpan1, and (c) delete a link from subSpan2. addSpan is selected first from any span (O(S) choices). subSpan1 is then selected from the extended local spans of addSpan (ie. subSpan1 $\in$ $loc^2Span[addSpan]$). The propagating conditions which lead to the use of the extended local spans in this situation are as follows: The link added to addSpan may provide an alternate restoration path for any span within its local area (locSpan[addSpan]). The set of spans which gained a restoration path from the addition to addSpan is called the gainSet, where gainSet is a subset of locSpan[addSpan]. Now consider a span within gainSet. This span has an increased number of possible restoration paths; therefore, one of its restoration paths may be surplus. Any surplus spans will be contained within locSpan[gainSet] = $loc^2Span[addSpan]$.

When selecting subSpan2, it is recognized that two previous operations have been performed on the network spares (subSpan1 and addSpan). Thus, a useful link for removal from subSpan2 might be found on any span within the extended local area of addSpan (by the same argument as for subSpan1), or in the local area of subSpan1. Therefore, the number of spans from which subSpan2 can be selected is $|loc^2span[addSpan]|+|locspan[subSpan1]|-intersection(|loc^2span[addSpan]|,|locspan[subSpan1]|)$. The most common situation is when both subSpan1 and subSpan2 are in the extended local area of addSpan, suggesting that the additional spare link on addSpan created multiple surplus links. However, occasionally subSpan2 can be outside the extended local area of addSpan, because it was made surplus by the change

on subSpan1. Again, this is a case in which a trap has appeared around which path selection order impacts k-shortest path flow.

Therefore, the extent of search is
$O(S \cdot ||loc^2span[addSpan]|| \cdot (||loc^2span[addSpan]|| + ||locspan[subSpan1]||)) = O(20 \cdot S \cdot S'^2) = O(S \cdot S'^2)$.

When recalculating the restorability of the network following all three operations, note that any one of the changes may affect the restorability of any span in their locality. Therefore, the restorability of all spans in the union of the localities of addSpan, subSpan1 and subSpan2 must be included in S_c for recalculation. However, these calculation sets can be reduced by performing restorability calculation in stages. If add1_sub2() recalculates R_n after adding a link to addSpan, it can recalculate R_n for subsequent operations on the network sparing within the localities of those subsequent changes. Thus, add1_sub2() will not reassess the spans in the locality of addSpan but not in the locality of subSpan1 for the subtraction from subSpan1. If add1_sub2() recalculates R_n after operations on addSpan and subSpan1, it can assess all of the subsequent possibilities for subSpan2 by recalculating $R_{s,i}$ only those spans in the locality of subSpan2. Therefore, at line 23 in Figure 7.23 where the maximum extent of search leads to a restorability calculation, S_c will only include the local spans of subSpan2. Therefore, for the add1_sub2() module, S_c is effectively O(S').

This implementation also includes a prescreening stage: If subSpan1 and subSpan2 are within the extended locality of addSpan, the restorability will usually only be intact if both subSpan1 and subSpan2 are individually removable without having an impact on restorability. This condition is encoded with the subtractSet set of spans. For this prescreening stage to miss an opportunity is a rare occurrence, limited to the simultaneous existence of two conditions: (a) Removing of subSpan1 does not maintain restorability; and (b) if subSpan2 and subSpan1 are simultaneously removed, restorability is maintained. This slight discrepancy is allowed because there is no error in the restorability calculations, only the failure to seize a rare opportunity for improvement. Moreover, this opportunity is somewhat of a wolf in sheep's clothing, in that a tighter network design in a topology for which path selection order can impact restorability is not desirable.

Note that the actual implementation of add1_sub2() which was tested in the current research did not incorporate the possibility of a subSpan2 which is not from within loc^2Span[addSpan]. Therefore, line 19 in this module was replaced with: "for all (subSpan2 in [subtractSet]) {." This change speeds up the procedure, and the penalty in performance is both small and non-crucial, in the same sense as above: Opportunity is lost, but network feasibility is not compromised.

```
procedure add1_sub2(networkRest, spanRest) {
1       success <- FALSE;
2       for all (addSpan in S) {

3               found <- FALSE;
4               spare[addSpan] <- spare[addSpan] + 1;
5               subtractSet <- NULL;

6               tempNetworkRest <- networkRest;
7               tempSpanRest <- spanRest;
8               addNetworkRest <- calc_rest(locSpans[addSpan],
tempNetworkRest,
                                tempSpanRest);
9               addSpanRest <- tempSpanRest;
10              for all (subSpan1 in loc2Span[addSpan]) {
11                      spare[subSpan1] <- spare[subSpan1] - 1;

12                      tempNetworkRest <- addNetworkRest;
13                      tempSpanRest <- addSpanRest;
14                      sub1NetworkRest <- calc_rest(locSpan[subSpan1],
                                tempNetworkRest, tempSpanRest);
15                      sub1SpanRest <- tempSpanRest;

16                      if (sub1NetworkRest >= networkRest)
17                              subtractSet <- subtractSet + subSpan1;
18                      else continue; /* choose a new subSpan1 */
19                      for all (subSpan2 in ([subtractSet] + locSpan[subSpan1] -
                        intersection(loc2Span[addSpan],locSpan[subSpan1]))) {
20                              spare[subSpan2] <- spare[subSpan2] - 1;

21                              tempNetworkRest <- sub1NetworkRest;
22                              tempSpanRest <- sub1SpanRest;
23                              calc_rest(locSpan[subSpan2], tempNetworkRest,
                                tempSpanRest);

24                              if (tempNetworkRest >= networkRest) {
25                                      spanRest <- tempSpanRest;
26                                      networkRest <- tempNetworkRest;
27                                      found <- TRUE;
28                                      success <- TRUE;
                        }
29                              if (found = TRUE) break; /* end inner for loop */
30                              spare[subSpan2] <- spare[subSpan2] + 1;
                }
31                      if (found = TRUE) break; /* end outer for loop */
32                      spare[subSpan1] <- spare[subSpan1] + 1;
        }
33              spare[addSpan] <- spare[addSpan] - 1;
  }
34      if (success = TRUE) return COMPLETE;
35      else return FAILED;
}
```

Figure 7.23 Add One and Remove Two Spare Links SLPA Module With Locality Considerations

7.2.5.5 Add Two Spare Links While Deleting Three Spare Links

The add2_sub3() procedure uses the same principles applied to add1_sub2(). Figure 7.24 presents an implementation of this procedure. The complexity analysis of this procedure works from the knowledge obtained in the add1_sub2() analysis.

In add2_sub3() there is one more level of nesting for the second addition of a spare link, plus one more level of nesting for the third removal of a spare link.

The second spare link is added to any network span (addSpan2) which shares an extended local span with the addSpan1 (ie. intersection(loc^2Span[addSpan1], loc^2Span[addSpan2]) <> NULL). Because this is a large number of spans, it is possible (even in large networks by today's standards) that addSpan2 will be taken from all S spans. Returning to the analogy of the previous section where the number of spans in an extended locality is described by the area of a circle of radius 2·r, the number of spans that have extended localities which intersect the extended locality of addSpan1 will be within a circle of radius 4r. Thus, the number of spans from which addSpan2 can be selected will be within four times the radius of a circle which represents locSpan[theSpan]. Thus, addSpan2 can be selected from O(min(16·S',S)) spans.

The subtract spans can then be chosen as in the add1_sub2() discussion:

subSpan1 $\in$ loc^2Span[addSpan1] U loc^2Span[addSpan2];

subSpan2 $\in$ loc^2Span[addSpan1] U loc^2Span[addSpan2] U locSpan[subSpan1]; and,

subSpan3 $\in$ loc^2Span[addSpan1] U loc^2Span[addSpan2] U locSpan[subSpan1] U locSpan[subSpan2].

Therefore, the extent of search for add2_sub3() is $O(S\cdot(16\cdot S')\cdot(2\cdot4\cdot S')\cdot(2\cdot4\cdot S'+S')\cdot(2\cdot4\cdot S'+2\cdot S')) = O(S\cdot11520\cdot S'^4) = O(S\cdot S'^4)$.

This analysis can be extended to the complexity of addn_subn+1. Each subsequent addSpan must intersect the extended locality of one of the previous addSpan's. Each subsequent subSpan must be from within the locality of an addSpan or a previous subSpan. Therefore, the extent of search for addn_subn+1 is: $O(S\cdot S'^{(2\cdot n)}\cdot(16(n-1)!)\cdot((4\cdot n+n)!) / ((4\cdot n-1)!))$. This factorial increase in complexity does not allow indefinite increases in n. In the network sizes considered here, any increase in n beyond 2 would likely result in checking all network spans and therefore an extent of search of $O(S^{2n+1})$.

add2_sub3() is the most expensive module of SLPA in terms of actual computation time and asymptotic complexity. In order to reduce the execution time (by reducing the constant factor of 11520) so that an add2_sub3() module is feasible, it was not actually implemented as presented here. Instead, addSpan2 was selected from within the locality of addSpan1, which represents the most probable case whereby the two spans can complement each other in the

```
procedure add2_sub3(networkRest, spanRest) {
1       success <- FALSE;

2       for all (addSpan1 in S) {
        for all (addSpan2 in S such that
                        intersection( loc2Span[addSpan1], loc2Span[addSpan2]) <> NULL) {

3               found <- FALSE;
4               spare[addSpan1] <- spare[addSpan1] + 1;
4               spare[addSpan2] <- spare[addSpan2] + 1;

5               tempNetworkRest <- networkRest;
6               tempSpanRest <- spanRest;
7               addNetworkRest <- calc_rest(locSpans[addSpan1] +
                        locSpans[addSpan2], tempNetworkRest, tempSpanRest);
8               addSpanRest <- tempSpanRest;

9               subtractSet <- NULL;

10              for all (subSpan1 in loc2Span[addSpan1]+loc2Span[addSpan2]) {

11                      spare[subSpan1] <- spare[subSpan1] - 1;

12                      tempNetworkRest <- addNetworkRest;
13                      tempSpanRest <- addSpanRest;
14                      sub1NetworkRest <- calc_rest(locSpan[subSpan1],
                                        tempNetworkRest, tempSpanRest);
15                      sub1SpanRest <- tempSpanRest;

16                      if (sub1NetworkRest >= networkRest)
17                              subtractSet <- subtractSet + subSpan1;
18                      else continue; /* choose a new subSpan1 */

                        subtractSet2 <- NULL;
19                      for all (subSpan2 in ([subtractSet] + locSpan[subSpan1] -

        intersection(loc2Span[addSpan1]+loc2Span[addSpan2],locSpan[subSpan1]))) {

20                              spare[subSpan2] <- spare[subSpan2] - 1;

21                              tempNetworkRest <- sub1NetworkRest;
22                              tempSpanRest <- sub1SpanRest;
23                              sub2NetworkRest <- calc_rest(locSpan[subSpan2],
tempNetworkRest,
                                        tempSpanRest);
                                sub2SpanRest <- tempSpanRest;

                                if (sub2NetworkRest >= networkRest)
17                                      subtractSet2 <- subtractSet2 + subSpan2;
                                else continue; /* choose a new subSpan2 */
```

Figure 7.24 (a) Add Two and Remove Three Spare Links SLPA Module (With Locality)

```
19                              for all (subSpan3 in ([subtractSet2] +
                                        locSpan[subSpan2] -
                                        intersection(loc2Span[addSpan1]
                                                + loc2Span[addSpan2],
                                                locSpan[subSpan2]))) {

20                                      spare[subSpan3] <- spare[subSpan3] - 1;

21                                      tempNetworkRest <- sub2NetworkRest;
22                                      tempSpanRest <- sub2SpanRest;
23                                      calc_rest(locSpan[subSpan3],
                                                tempNetworkRest, tempSpanRest);
24                                      if (tempNetworkRest >= networkRest) {
25                                              spanRest <- tempSpanRest;
26                                              networkRest <- tempNetworkRest;
27                                              found <- TRUE;
28                                              success <- TRUE;
                                        }
29                                      if (found = TRUE) break; /* end inner for loop */
30                                      spare[subSpan3] <- spare[subSpan3] + 1;
                                }
                                if (found = TRUE) break; /* end middle for loop */
                                spare[subSpan2] <- spare[subSpan2] + 1;
                        }
31                      if (found = TRUE) break; /* end outer for loop */
32                      spare[subSpan1] <- spare[subSpan1] + 1;
                }
33              spare[addSpan] <- spare[addSpan] - 1;
        }
34      if (success = TRUE) return COMPLETE;
35      else return FAILED;
}
```

Figure 7.24 (b) Add Two and Remove Three Spare Links SLPA Module (With Locality)

same restoration path. This assumption is supported by the fact that the add1_sub2() module is known to have failed before the execution of this module. Thus, the execution time results presented here reflect an extent of search of $O(S \cdot S' \cdot (2 \cdot S') \cdot (3 \cdot S') \cdot (4 \cdot S')) = O(S \cdot S'^{4} \cdot 4!) = O(S \cdot S'^{4})$. Thus, the extent of search for the case of addn_subn+1 when presented similarly to add2_sub3() is $O(S \cdot S'^{(2 \cdot n)} \cdot (2 \cdot n)!)$.

As in add1_sub2(), the calculation of R_n is performed in stages and, therefore, the calculation at the deepest nesting will only require recalculation for the S' spans in the locality of subSpan3. Therefore, $S_c = O(S')$. Also, for the general case of addn_subn+1, $S_c = O(S')$.

7.3 Complexity Analysis of SLPA

An experiment described in the previous section identified the benefits of reducing the complexity of the restorability calculation; there, the most optimized form of dijk_calc_rest() accounted for 99.9% of the execution time of SLPA. Hence, four factors contribute to the

complexity of SLPA: (1) the complexity of path-table preparation, (2) the total number of iterations in FS and DS, (3) the extent of search at each iteration, and (4) the complexity of calculating R_n. The complexity of SLPA can then be expressed as the greater of the complexity of (1) and the product of the complexities of (2) through (4).

Only the path-table implementation of SLPA uses the path-table; other versions all use metaDijkstra for path-finding. The total number of (FS+DT) steps is approximately the same for all versions of SLPA on a given network. The extent of search at each step is also the same for all versions except for SLPA Short which omits add2_sub3(). The required time for calculating R_n differs markedly between metaDijkstra and path-table implementations.

7.3.1 Worst Case Analysis

In Section 7.2, the asymptotic complexity of the preconditioning stage was already presented for the path-table implementation ($O(S \cdot d_{max}^{RPL})$) and is negligible in the Dijkstra implementation.

The second factor in overall SLPA complexity is the number of steps in FS and DT phases. The FS phase cannot require more than $W = \sum_{i=1}^{S} w_i$ steps, because at least one additional working link is restorable after each step. The DT phase also cannot require more than W steps, because at most W spares (the upper-bound redundancy for mesh-restoration is 100%) will have been added by FS, and DT will not remove them all. The worst case, then, for the number of steps is O(W).

The worst case extent of search was shown in Section 7.2.5.5 to occur for the addn_subn+1 module which has $O(S \cdot S'_{max}{}^{2 \cdot n} \cdot (2 \cdot n)!)$. Here, S'_{max} is the maximum number of spans in a locality and therefore represents worst case complexities of the analyses in Section 7.2. (In Section 4.1.4 it was shown that $S'_{max} = O(\min(S, d_{max}{}^2 \cdot RPL^2/8))$ spans or $N'_{max} = O(\min(S, d_{max} \cdot RPL^2))$nodes.) Therefore, in networks where the values of d and RPL are large compared to the network size, the extent of search in a given step becomes $O(S^{(2 \cdot n+1)})$, representing a full combinatorial search of the network for groups of spans to be operated upon. (In this case, the implementations using localities (Section 7.2.5) reduce to the basic implementations of Section 7.1.) For example, add2_sub3() operates on 5 spans, thus requiring $O(S \cdot S'_{max}{}^4)$ alternatives to be examined. The worst case complexity is dominated by the module which operates on the largest number of spans-- either add1_sub2() in SLPA Short or add2_sub3() otherwise.

For both the metaDijkstra and path-table implementations, calculating R_n for a single span operation (alternative within the search) requires calculating $R_{s,i}$ for $S_c = O(S'_{max})$ spans. The complexity of calculating $R_{s,i}$ differs for the path-table and metaDijkstra implementations. For

the path-table, the worst case complexity of calculating $R_{s,i}$ involves a search of the entire table, which is $O(S \cdot d_{max}^{RPL})$. For metaDijkstra, the worst case for calculating $R_{s,i}$ is $O(w_i \cdot N'_{max} \cdot \log(N'_{max}))$ for one alternative. Therefore, the worst case complexity for calculation of R_n is $O(S \cdot S'_{max} \cdot d_{max}^{RPL})$ for SLPA Path-table and $O(W' \cdot N'_{max} \cdot \log(N'_{max}))$ and SLPA Dijkstra, where $W' = \sum_{i=1}^{S'} w_i \leq W$.

The overall worst case complexity for SLPA using metaDijkstra, then, is $O(W \cdot W' \cdot S \cdot S'^4_{max} \cdot N'_{max} \cdot \log(N'_{max}))$, where $S'_{max} \leq S$, $N'_{max} \leq N$, and $W' \leq W$. The overall worst case for the path-table implementation is $O(W \cdot S^2 \cdot S'^5_{max} \cdot d_{max}^{RPL})$.

If d and RPL are considered as constants, N', and S' are also constants. This is a reasonable assumption in telecommunication transport networks of today where RPL is generally less than 10 and d ranges from 2 to 5. (S' and N' have been expressed in terms of d and RPL.) W' does not disappear though unless the number of working links per span is constant. This reduces the worst case complexity to $O(W \cdot S^2)$ for path-table and $O(W^2 \cdot S)$ for metaDijkstra.

7.3.2 Experimental Results

The expected size of the path-table is $E(S \cdot RPL \cdot d^{RPL})$. Therefore, with RPL fixed at 10, the constant d_{avg} curves of Figure 7.25 are expected to be E(S) = E(N) ($S = d_{avg} \cdot N/2$). The graph shows nearly linear relationships for d_{avg}=3 and 4, but only one data point could be obtained at d_{avg}=5. The strong dependence on d_{avg} and RPL may restrict use of the path-table variation of SLPA to $2 \leq d_{avg} \leq 4$ networks. In Figure 7.26, the size of the path-table is fitted with a quadratic curve for the d_{avg}=4 data. The curve fit is correlated with the data to R^2 is 93%. Thus, for this data it appears that $E(S^2)$ is more indicative.

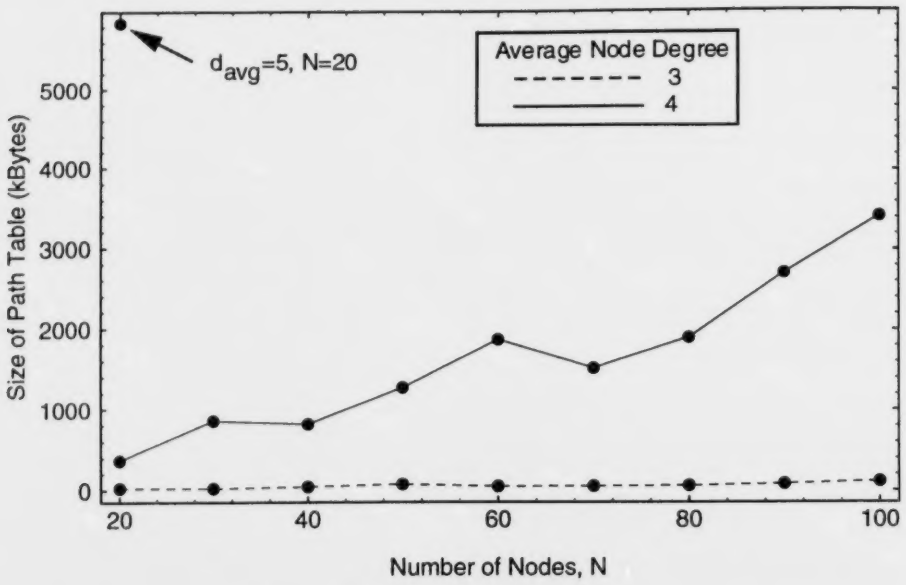


Figure 7.25 Path-Table Size for SLPA

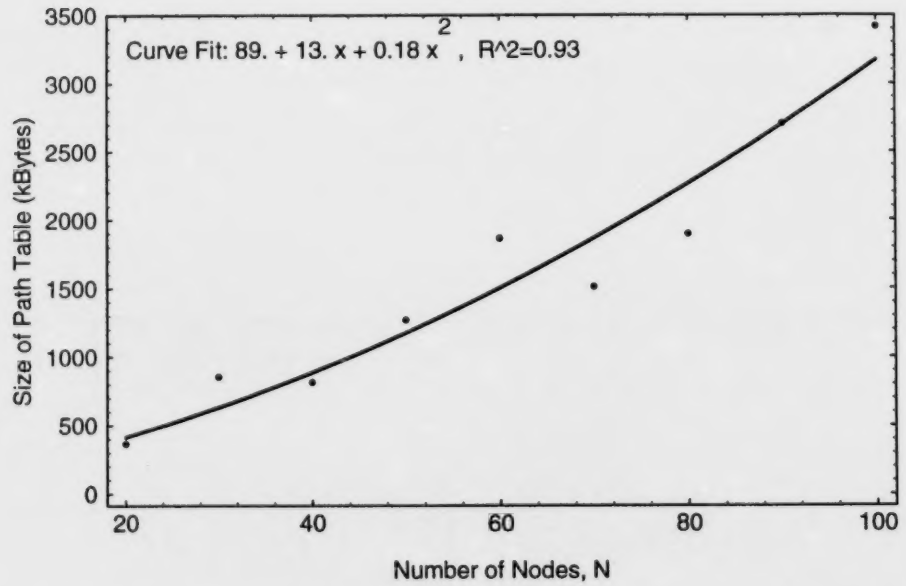


Figure 7.26 Quadratic Fit to Path-Table Data for Degree 4 Networks

The worst-case analysis predicts that the total number of steps to complete a design is a linear function of W. The graph in Figure 7.27 is a plot of the number of steps taken by SLPA (FS+DT) for each run, plotted against the number of working links in the network. This graph confirms that the number of steps is well approximated as E(W).

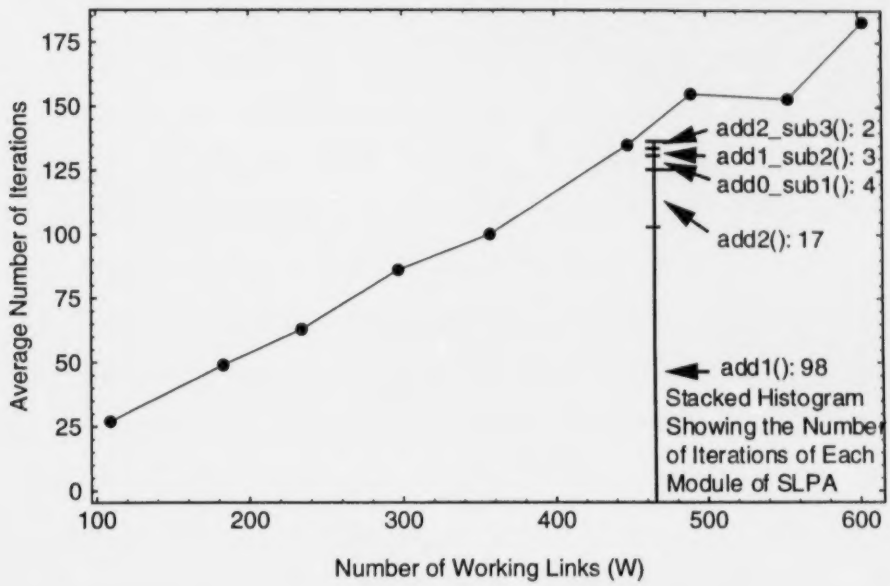


Figure 7.27 Complexity of Average Total Number of Steps of SLPA Executions (d_{avg}=4)

The extent of search at each step can be examined via the data in Figure 7.28. This is a plot of the number of calculations of R_n (there will be one call per alternative within the search) as a function of the number of spans in the network. The lower line, SLPA Short, increases linearly with S at a small slope; as predicted by the analysis, which says the dependency should be $E(S \cdot (S')^{n-1})$. Because S' is effectively a constant, we observe a complexity of E(S). The difference between the upper line and the lower one is that the path table version is using add2_sub3(). The higher variance of this line is reasonable since each point represents the addition of only 1-3 calls to this very computationally expensive module and the only network which required 3 calls to add2_sub3() was the 180 span network. Again, the the overall trend is best characterized as E(S).

Figure 7.29 shows the cost of evaluating R_n for one alternative as a function of the size of the network. With the use of localities, worst-case analysis predicts metaDijkstra to be E(W) because the factor $N' \cdot \log(N')$ is a constant; experiments confirm this. SLPA Path-table gets some help from the use of localities, because only spans in the locality of a change need to be recalculated. However, the use of localities does not impact the size of the path table- one

search could take up to S steps. Thus, we expect a complexity of E(S). Although the slope of the line is very shallow, Figure 2.29 confirms that E(S) time is required for path-table searches.

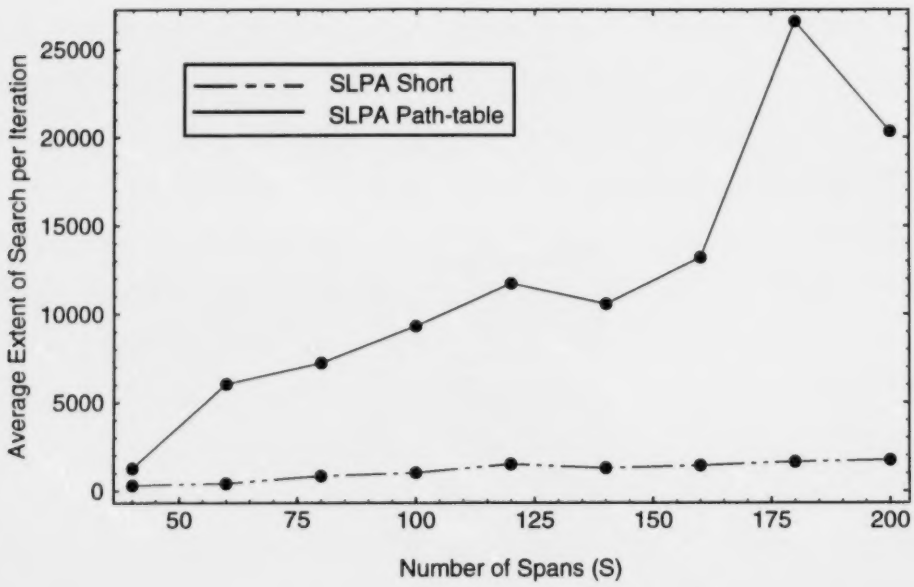


Figure 7.28 Complexity of Average Extent of Search of Each Step of SLPA (d_{avg}=4)

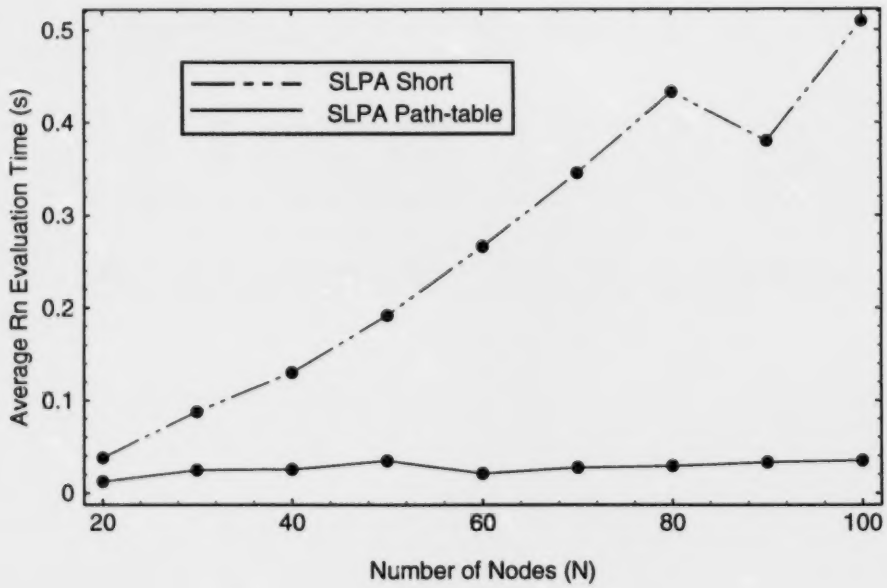


Figure 7.29 Average Time Complexity of $R_{s,i}$ Calculation (d_{avg}=4)

Overall, this gives an average-case complexity of $E(W^2 \cdot S)$ for the metaDijkstra implementation, and $E(W \cdot S^2)$ for the path-table implementation.

Figures 7.30 to 7.32 present an overview of these results. In the network designs used here, the working capacities are random, each with a uniform probability over the same range. Thus, effectively, O(S) = O(W) for the data presented here. Also, for a constant d_{avg}, O(S)=O(N). Therefore, the overall execution time curves, plotted against N, can be fitted with cubic curves to confirm the $E(W^2 \cdot S)$ and $E(W \cdot S^2)$ complexities. At the same time, these curves can provide the overall constant multiplier for each method with d_{avg}=4. Thus, figures 7.30, 7.31, and 7.32 show the execution time data collected for SLPA Path-table, SLPA Dijkstra, and SLPA Dijkstra Short, respectively. These times were recorded on a SUN SPARC 2™ RISC-based diskless workstation with 16Mb main memory and 48 Mb swap space. The execution times observed in SLPA Dijkstra were an order of magnitude higher than those of the other two SLPA variations. However, all three variations of SLPA exhibit nearly cubic increases in execution time with respect to network size. The correlation between cubic curve fits and the observed times were R^2 = 0.92, 0.99, and 0.96 for SLPA Path-table, SLPA Dijkstra, and SLPA Dijkstra Short, respectively. From these cubic curve fits, the constant multipliers for large N values (ie., the coefficient of the N^3 term) are 0.18, 3.58, and 0.08 for SLPA Path-table, SLPA Dijkstra, and SLPA Short, respectively. Therefore, the lowest expected computation time for large networks will be $0.08 \cdot N^3$ seconds. However, the path-table implementation can achieve nearly the same time, $0.18 \cdot N^3$, while including the add2_sub3() process.

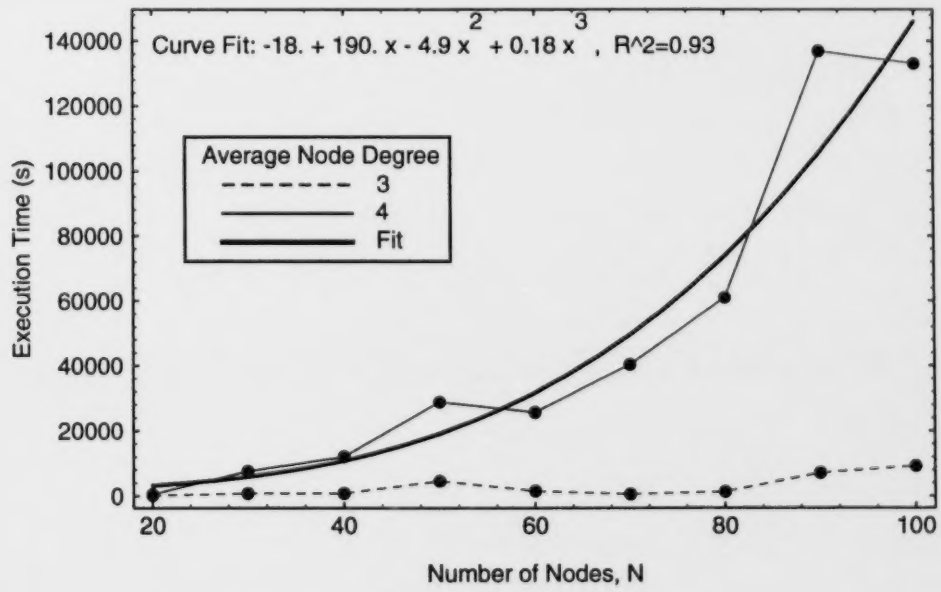


Figure 7.30 Execution Times for SLPA Path-Table

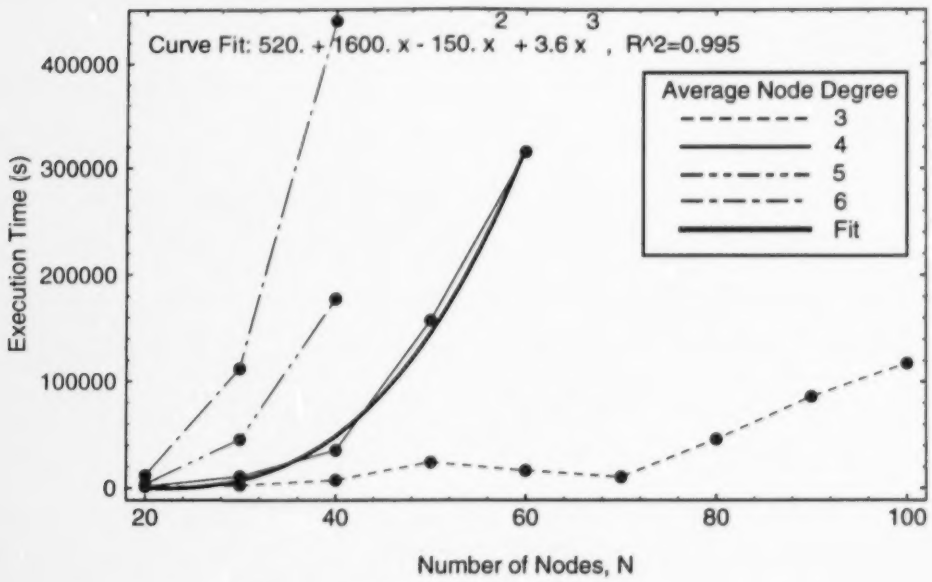


Figure 7.31 Execution Time for SLPA Dijkstra

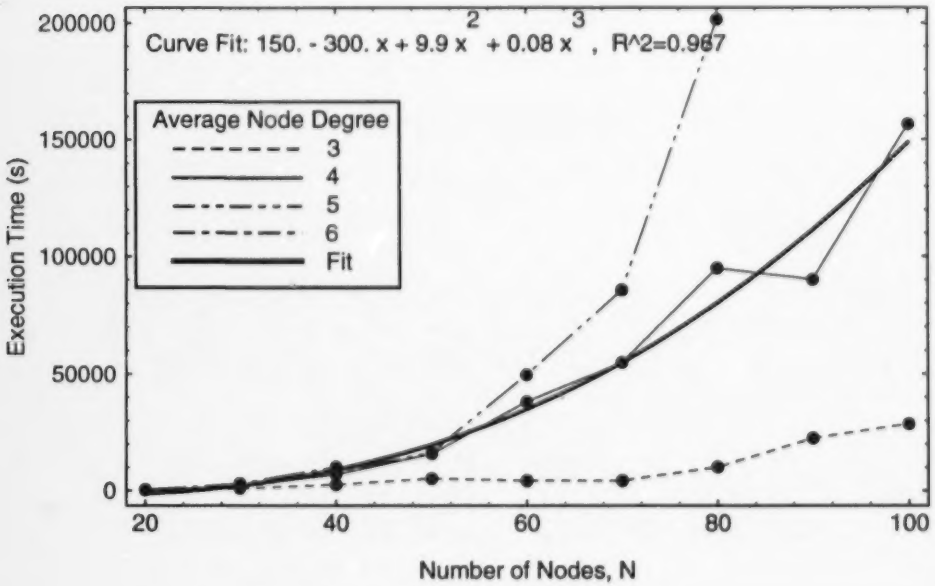


Figure 7.32 Execution Time for SLPA Short

7.4 Restoration Type and Restorability

All of the implementations of SLPA operate strictly within k-shortest paths restoration regimes, an important observation when it is assumed that fast restoration algorithms will generally be of this nature as well.

The discussion of Section 4.2 observed that the k-shortest paths flow is dependant on the order of path selection when faced with multiple paths of identical length. Therefore, even though a k-shortest paths algorithm was used to design the network, it is not true that a different type of k-shortest paths algorithm will necessarily be able to achieve full restoration. The SLPA method of SCP can accommodate any restoration algorithm for which the path selection characteristics are known. This is accomplished through using the restoration algorithm itself as the module which assesses the restorability of a span (replacing metaDijkstra() or span_rest_pt()). But the module used for assessing span restorability must execute very quickly in order for SLPA to design a feasible network -- metaDijkstra() and span_rest_pt() are both optimized for fast execution.

The path-table implementation (span_rest_pt()) allows no freedom within the order of selection of equivalent length restoration paths, a rigidity which reflects the sorting required in the preprocessing stage where the path-table is developed. The paths are selected such that the most highly utilized spans can be assessed first; therefore, when choices exist between paths of the same length, the preferred path is the one which contains the more commonly involved spans. The path-table implementation cannot switch to a k-shortest physical restoration path length selection criterion. Such a scheme would result in few paths of equivalent length, so path-table reductions and sorting would not be possible -- the path-table would be no faster than a Dijkstra implementation.

The Dijkstra implementation (metaDijkstra()) allows freedom in the path selection criterion. Here, three path selection schemes were implemented: logical path length (LOG), physical path length (PHYS), and logical path length with arbitration of common lengths by physical path length (LOG+PHYS). Note that LOG was the only scheme allowed in the path-table implementation.

In metaDijkstra(), LOG performs the same function (for which it was selected in the path-table implementation) of providing the best execution speed. This advantage results from its ability to select multiple restoration routes after a single call to dijkstra() (see Section 4.1.6.2). In the PHYS scheme, no two routes will be of the same length because span lengths are real-valued; therefore, each route must be identified separately through calls to dijkstra(). In the LOG+PHYS scheme, the short-cut introduced into metaDijkstra which finds multiple routes of the same logical length in one call to Dijkstra cannot be used. This short-cut does not find all routes of the same logical length, but only those which are span disjoint from each other. Therefore, this short-cut may find multiple paths of the same logical length, while omitting another path of the same logical length which has a lower physical length. Therefore, in the LOG+PHYS scheme, only single routes can be accepted in each call which metadijkstra() makes to Dijkstra().

Does any one of these path selection schemes provide a network design which is more restorable by other schemes, thereby providing a better design for restoration by an algorithm with unknown or undefinable characteristics? This question is addressed by Table 7.1 for the schemes identified above, with a known distributed parallel restoration algorithm (SHN - [Grov87]). The maximum table entry of 17 corresponds to a network restorability of 98.8%, suggesting that the mismatch between different restoration schemes is generally less than 2%.

Table 7.1 Number of Unrestorable Links (Out of 1443 possible) when a Network Design Based on One k-Shortest Paths Restoration Scheme is Restored by Another

Design Scheme	Restoration Scheme				
	Path-Table	LOG	PHYS	LOG+PHYS	SHN
Path-Table	0	6	5	5	5
LOG	4	0	0	0	1
PHYS	7	10	0	5	2
LOG+PHYS	3	2	17	0	1

The values of 10 and 17 indicate a situation which should be avoided in network design: matching the PHYS design tool with the LOG restoration method, or vice-versa. In general, if a network is designed by a LOG design tool, then the shortest available physical path, which will be found by a PHYS restoration method, may be longer than RPL even though a length RPL path is available. The restoration method behaves as though it selects paths physically, but it only selects paths within a specified logical length (RPL). If the restoration method does not concern itself with logical lengths at all and seeks only paths which are physically short, this mismatch will not be as extreme.

The LOG-based network design demonstrates the greatest compatibility with all forms of restoration algorithms. It also supplies the largest number of links, which is probably the reason for its advantage over LOG+PHYS. However, any method which selects paths primarily based on logical length will facilitate high restorability by SHN and other k-shortest path restoration algorithms.

These results suggest that before any network design is employed, it should be tested using the actual restoration algorithm. Then minor modifications can be incorporated to compensate for shortcomings in restorability of the design. The differences found above are, however, wholly acceptable for planning studies where significant differences, 10% to 300%, between network restoration alternative technologies are being assessed.

7.5 Transmission System Capacity Modularity

Because system modularity is an integral part of the SLPA algorithm, SLPA provides the same level of optimality for networks defined modularly as those to which links can be added one at a time. In the foregoing discussion, all references to a single link operation (subtraction or

addition) are replaced by system (module) operations. The SLPA has not been implemented to accommodate variable modularities, but this would only require an extra test during the FS phase to ensure that the smallest modules are added before considering larger modules.

7.6 Summary

SLPA was implemented in two main forms: SLPA Path-table and SLPA Dijkstra. An alternate version, SLPA Short, is equivalent to SLPA Dijkstra, but it does not include the final module in design tightening (add2_sub3()).

The complexity of the SLPA is in **P** with respect to network size. However, the path-table implementation is exponential in RPL in both time and memory space.

SLPA provides for k-shortest paths restoration characteristics, the same type of restoration capacity which is anticipated for fast restoration [Grov89]. It can include system modularity requirements during the synthesis of the network.

Chapter 8 compares SLPA to ICH, as introduced in Chapter 6, and describes SLPA's characteristics of network growth, over-restorability and ease of implementation.

8 Comparison of SCP Heuristics

This chapter presents a direct comparison of ICH, ICH RPL, SLPA Path-table, SLPA Dijkstra and SLPA Dijkstra-short. Chapter 4 introduced the basis of comparison. For many of the areas of comparison, Chapters 6 and 7 discuss merits of specific heuristic algorithms. The final section of this chapter presents recommendations for selection of SCP algorithms.

8.1 Execution Time

Figure 8.1 presents charts of the fully restorable network designs obtained by ICH and the three SLPA variations. All SCP experiments were executed on a SUN SPARC 2™ RISC-based computing platform with execution speed of 28.5 MIPS, from compiled "C" language source code. Diskless workstations were used with 16 Mb main memory and 48 Mb swap space. One criterion for our study was to limit run-time to one day; however, the charts include data for some runs up to four days in time where this was feasible. The white squares in Figure 8.1 are execution cases, generally large ($N \cdot d_{avg}$), that did not complete before four days of execution. All results are for RPL=10 which is a higher level than most practical networks require.

SLPA Dijkstra completed designs for 15 of the networks of less than 50 nodes, or nodal degree 3 within one day. The SLPA Short algorithm provided designs for 24 of the 36 network trials within one day. The SLPA Path-table implementation executed the most quickly for all networks at d_{avg}=3. However, the space requirements increase with the average node degree so rapidly that our computing platform could not provide the storage to keep the whole path-table in memory for most highly connected networks. The largest observed path-table occupied 5.8 Mbytes, for the N=20, d_{avg}=5 network.

With ICH, 16 of the network designs completed in less than one day. The networks on which ICH exhibited very long run times were not as systematically related to N and d as for SLPA. For example, the relatively small N=50, d_{avg}=4 network did not complete in over four days, whereas the N=60 trial network at the same d ran in under one day. This reflects the greater dependence of ICH on individual network details, due to its LP-based formulation. This kind of behavior is attributed to an LP data set for which the LP execution time exhibits its exponential potential. Apparently, knowing the input network size is not enough to predict whether ICH will succeed within a reasonable time-limit.

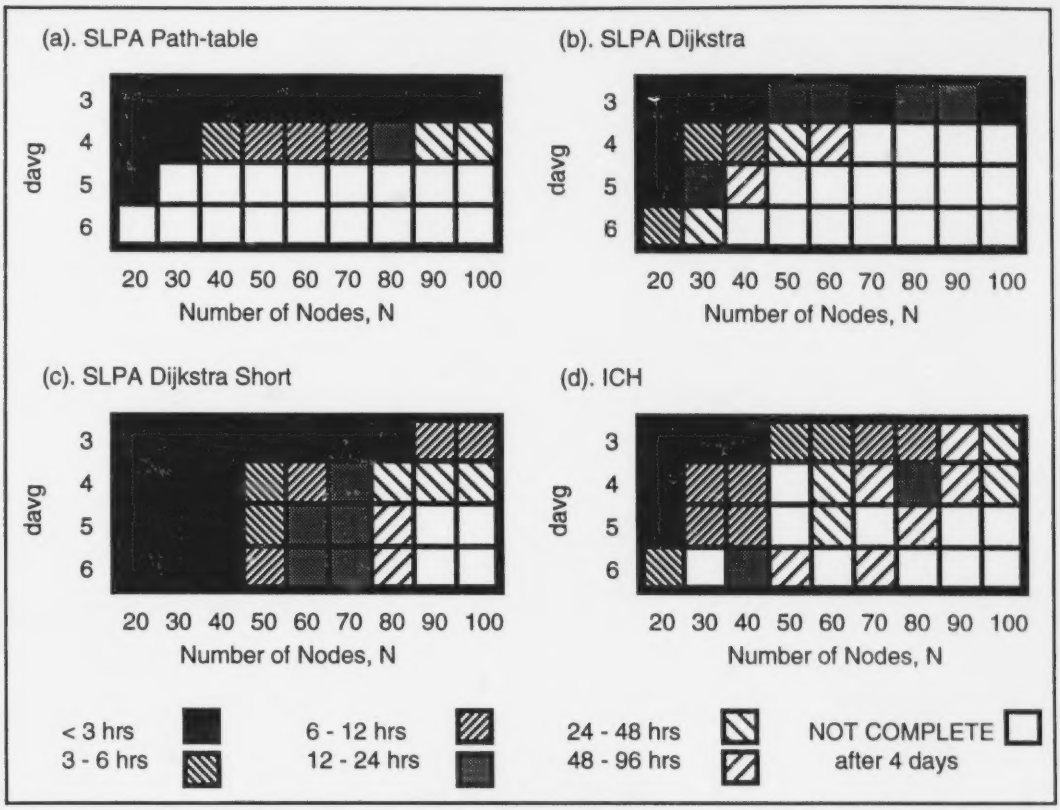


Figure 8.1 Execution Times for Four SCP Heuristics for 36 Different Trial Designs

Figure 8.2 shows the execution times for the d_{avg}=4 columns of Figure 8.1. These curves more clearly show trends in execution times as N increases. The SLPA execution times increase gradually and smoothly, showing a strong correlation to network size. ICH execution time increases and decreases significantly as the network size increases steadily. The 100 node network took 55% less time to design than the 90 node network and the 50 node (d_{avg}=4) network did not complete within the cutoff of 4 days of execution time.

In chapters 6 and 7, the data of Figure 8.2 were fitted with polynomial curves. The execution times were found to be $E(4.04 \cdot N^2 \cdot (\text{number of iterations}))$ for ICH, $E(0.18 \cdot N^3)$ for SLPA Path-table, and $E(0.08 \cdot N^3)$ for SLPA Short. Although ICH has the lowest average-case time complexity, some designs required exponential time to complete, such as the 50-node network of Figure 8.2 that had not completed in over 500 000 seconds. The number of iterations of ICH with d_{avg}=4 was observed to be nearly constant at 5. Therefore, a cross-over point in network size can be obtained where ICH is expected to execute more quickly than SLPA Path-table. This is at:

$$5 \cdot 4.04 \cdot N^2 \leq 0.18 \cdot N^3; \qquad N > 112.$$

Therefore, for d_{avg}=4 networks with more than 112 nodes, ICH is expected to execute more quickly than SLPA Path-table, on average.

However, the investigations in Chapter 6 showed that ICH has worst-case time complexity of exponential order, because the number of iterations can approach the maximum number of cutsets in the limit, and the execution time of the Simplex LP can be exponential. An LP can be designed which has worst-case complexity in P with respect to the constraint set size, but such an LP has not yet been discovered with average-case execution times which are feasible for use on non-trivial problems [Schr86]. Conversely, the SLPA algorithm has worst-case time complexity which is polynomial. Therefore, the SLPA method will more reliably provide network designs even when N>112 and the average-case executions may favor ICH.

For the network sizes considered in Figure 8.2, based on eight d_{avg}=4 trials, SLPA Short and SLPA Path-table designs executed in 29% of the time required for ICH designs on the same networks. The largest difference occurs, for the 20 node network, where SLPA Path-table executed in 14% of ICH's run time. Also, for the 70 node network, SLPA Path-table executed in 19% of ICH's run time. The smallest difference occurs, for the 100 node network, where SLPA Path-table executed in 89% of ICH's run time. d_{avg}=4 is considered appropriate for this comparison because most real transport networks are expected to be bounded reasonably tightly in $2<d\leq4$.

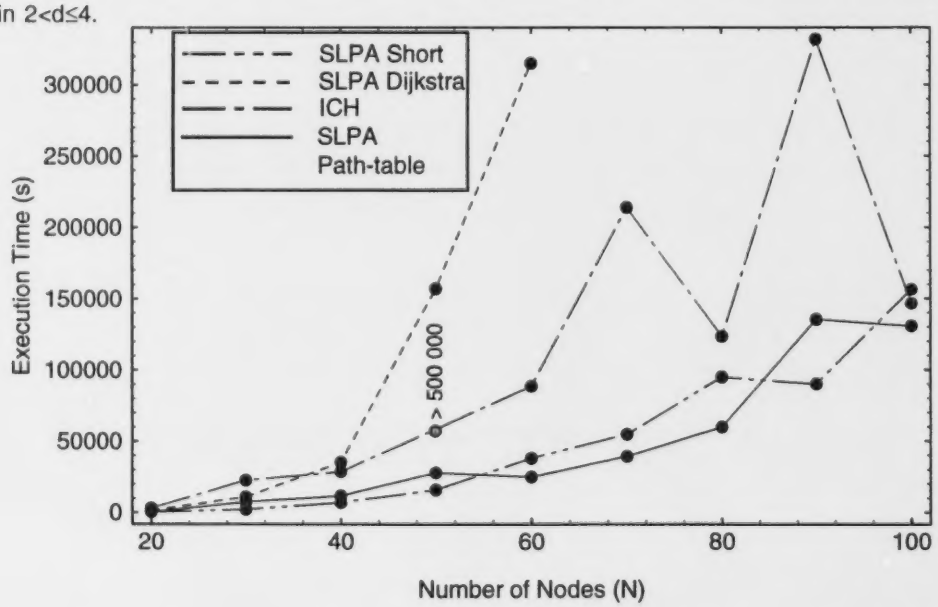


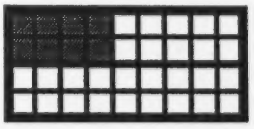
Figure 8.2 Comparison of Execution Times of ICH and Three SLPA Heuristics for Design of d_{avg}=4 Study Networks

8.2 Restorability

Ultimately, the network design must be compatible with the field-deployed restoration algorithm that is used. Thus, a selected restoration algorithm best evaluates each heuristic for restorability in operational use. The restoration algorithms which operate most efficiently on mesh networks are better approximated by k-shortest paths flows than by max-flows (see Section 4.2). This generalization implies that real-time restoration will probably select paths on a basis more similar to k-shortest paths than to max-flow.

Here, experiments exploit the SHN™ protocol, which has a net result very similar to k-shortest paths [Grov89], to test the restorability of the designs generated. Table 8.1 summarizes the results of these experiments. The icons to the right of the tables in this chapter indicate the set of network designs represented by the data. For Table 8.1, the data includes the degree 3 and 4 networks with nodes numbering 20, 30, 40, and 50. As expected, the k-shortest path algorithms do not attain 100% restorability because of a dependence on path selection order, but the number of unrestorable working links is small in all cases. For SLPA Path-table, there are 5 of 1443 unrestorable working links in the 8 network designs. For SLPA Short, there is 1 of 1443 unrestorable working link. SLPA Short has higher restorability in general because network design without the add2_sub3() SLPA module allows more spare capacity and, therefore, more potential restoration routes. The ICH RPL network designs did not achieve the levels of restorability of the SLPA designs. SHN operating on networks designed by ICH RPL left 52 working links unrestorable out of 1443. The ICH RPL data demonstrates the observation of [Grov89] that SHN can often out-perform k-shortest paths algorithms because of its highly parallel nature.

Table 8.1 Number of Unrestorable Links (of 1443 Possible) When Network Designs are Restored by k-Shortest Paths Schemes



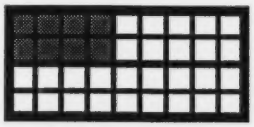
Design Heuristic	Restoration Scheme		
	Path Table	LOG	SHN™
SLPA PT	0	6	5
SLPA Short	4	0	1
ICH RPL	80	82	52

Table 8.2 compares statistics for the restorability of the network designs by SHN . Although it sought 100% restorability, ICH RPL achieved only 96.9% restorability due to the fact that SHN cannot fully exploit its inherent max-flow configuration. The SLPA network designs were also less than 100% restorable by SHN, because of the differences between k-shortest

paths selection orders; however, with SHN restoration, the SLPA Short implementation is 99.9% restorable on average

Table 8.2 Restorability Statistics for Restoration by SHN

Design Heuristic	Mean	Variance	Median
SLPA PT	99.7%	0.11%	99.8%
SLPA Short	99.9%	0.01%	100%
ICH RPL	96.6%	11.49%	97.7%



As discussed in Chapter 7, the SLPA algorithm can incorporate a final FS phase which tests restorability using the field-deployed restoration algorithm itself, and can therefore always achieve strictly 100% restorability. No such capabilities have been discovered for the ICH method yet.

8.3 Capacity Efficiency

The previous section showed that the various algorithms do not provide completely equivalent levels of restorability because they have been designed to satisfy slightly different restoration criteria. These inherent differences are also reflected in the capacity efficiency of the designs. The ICH designs inherently imply max-flow restoration and, therefore, should require somewhat less redundant capacity than the SLPA algorithms.

Figure 8.3 makes a comparison of the total redundant capacity contained in SLPA and ICH RPL designs. The lower bound is based on the real valued outputs from the LP of the final step of the ICH RPL design. This corresponds to hypothetical real-valued max-flow type of restoration in a network of artificial real-valued span capacities. An actual capacity placement which matches this bound while using integer capacities and k-shortest paths restoration will not exist in general. The various heuristics provide designs which differ by only a few percent in terms of total spare capacity above this bound.

ICH RPL network designs strictly require less capacity than SLPA because they inherently assume max-flow restoration characteristics rather than k-shortest paths restoration (as in SLPA). ICH RPL designs required 440 spare links to restore 823 working links in total over the N=20 to 50, d_{avg}=3 and 4 study networks. The SLPA Path-table designs required (in total) 24 more spare links (5%) than ICH RPL. The SLPA Short design uses 9 more spare links (2%) than the corresponding SLPA Path-table design because of the absence of the "add2_sub3()" module.

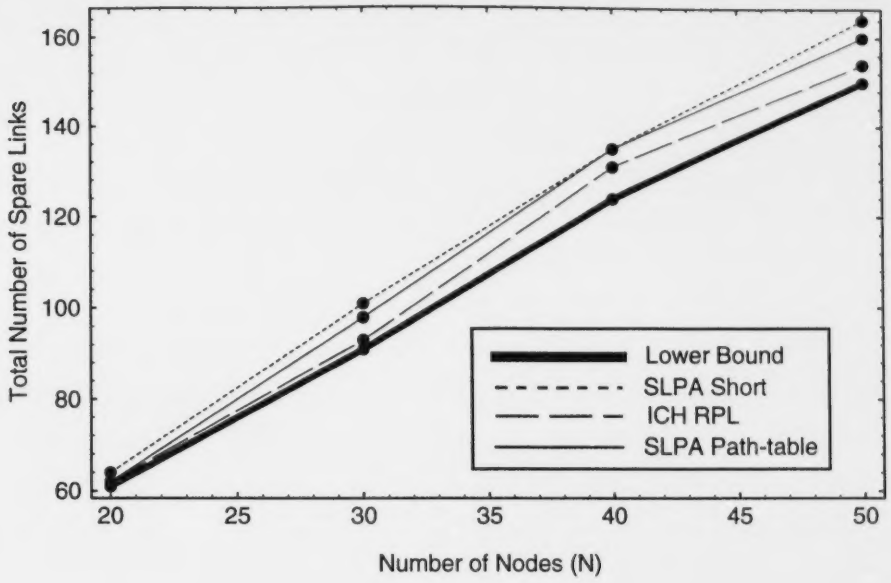
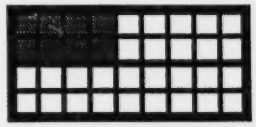


Figure 8.3 Comparison of Total Spare Capacity Requirements of d_{avg}=4 Study Networks Using ICH RPL and Two SLPA Heuristics for Design

Table 8.3 contains a statistical comparison of the redundancy requirements of the networks generated by the various design heuristics. In these calculations, the redundancy of each design was determined as the ratio between total spare capacity and total working capacity. The statistics are based upon this normalized redundancy value in order to equally weight the designs even though the network sizes are different. The average redundancy is approximately 5% less for the ICH RPL design compared to the SLPA Short design.

Table 8.3 Redundancy Statistics ($\sum$ Spare Capacity / $\sum$ Working Capacity)

Heuristic	Mean Redundancy	Variance
ICH RPL	0.6845	0.0249
SLPA PT	0.7079	0.0270
SLPA Short	0.7216	0.0276
Lower Bound	0.6739	0.0264



8.4 Accommodation of Network Growth

Both ground-up and incremental growth accommodation were implemented and tested, in ICH and SLPA, for effectiveness in accommodating growth in the 50 node, degree 4 study network. In each of 10 trials, one end-to-end working path was added to the network (shortest path routed) between two nodes selected at random. The network's SCP was then updated to

re-establish full restorability. Of interest in these trials is the amount of network rearrangement required by each method to regain a fully restorable state. Table 8.4 presents these results.

ICH and SLPA perform incremental growth accommodation by solving a smaller SCP problem in a network of the same span topology but in which $w_j > 0$ only where non-restorable spans exist in the full network due to growth. The spare links to realize full restorability in the subnetwork design are then added to the full network.

Use of ICH in the ground-up mode means running the whole SCP with the new network input file comprised of the grown working span capacities to be protected by a new SCP solution. For SLPA, ground-up design means (here) that FS and DT phases were repeated using the prior network state (after growth) as the new starting point. Because DT can remove and redistribute capacity to an arbitrary degree while satisfying the restorability target, this is considered functionally equivalent to a redesign in which SLPA was literally re-run from the one spare per span starting point. However, this method provides the beneficial bias towards an SCP solution which favors the capacity placements which already exist.

The SLPA ground-up redesign removed capacity from an average of 1.7% of the total spans (1.7 spans) and added capacity to an average of 5.7 spans. SLPA incremental growth accommodation increased capacity on 4.6 spans (decreasing none). In both approaches to growth accommodation, SLPA execution times were of the same order as the original design (approximately 2 hours for ground-up and 0.5 hours for incremental growth accommodation in this network with RPL=10). This reflects dominance of the DT phase which both ground-up and incremental modes of operation require.

Table 8.4 shows that ICH ground-up redesign resulted in changes to more spans than SLPA. ICH ground-up accommodation removed capacity from an average of 11.9% of the 100 spans (11.9 spans) during each growth trial. It increased an additional 16.9 spans on average during each trial. This is a considerable level of churn (29 changes on average to accommodate survivability for one additional working path). This suggests that ICH may be too sensitive in practice to totally re-run as a response to growth. With incremental growth accommodation, ICH requires considerably fewer changes to the network, as seen in the fourth row of Table 8.4. Here, incremental accommodation added capacity to 5.3 spans on average for the additional 3.8 working links in each growth trial. The execution time for ICH was between 5 and 24 seconds for incremental growth accommodation, as opposed to 3 to 10 hours in ground-up network redesign, reflecting the size of the smaller SCP problem incremental growth accommodation solves.

Table 8.4 Growth Accommodation: Average Effects for One Random Path Addition (N=50, davg=4 Study Network, 10 Path Addition Trials)

	SCP Heuristic	spans with reduced capacity	spans with increased capacity	working links added	design update execution time
ground-up	SLPA	1.7	5.7	3.6	2 hours
	ICH	11.9	16.9	3.7	3-10 hours
incremental	SLPA	-	4.6	3.6	0.5 hours
	ICH	-	5.3	3.8	5-24 seconds

In summary, both ICH and SLPA can accommodate network growth either by running incrementally or in ground-up redesign. However, when the heuristics perform ground-up growth accommodation, ICH designs require significantly more changes to the network than SLPA. The fastest method is to use ICH in incremental mode, but SLPA (incremental) still produces fewer changes to the network.

8.6 Over-Restorability

Over-restorability (5.3.6) side-effects for individual spans resulting from the synthesis of a fully restorable overall network were not investigated in chapters 6 and 7 because target over-restorability cannot be easily identified. Moreover, different implementations of a given algorithm are expected to provide over-restorabilities of the same magnitude. However, when the over-restorability of a network design from ICH and a network design from SLPA are compared, it may provide a measure of design margin beyond the simple restorability figure. Even when no restorability difference exists between network designs, a higher over-restorability can result in a lower requirement for the addition of spare capacity to accommodate network growth and less sensitivity to errors in deployed network state vis-à-vis the ideal design architecture.

ICH RPL, SLPA Path-table and SLPA Short all have similar over-restorabilities, as shown in Table 8.5. But, surprisingly, design methods which provide the most total capacity do not necessarily provide the highest over-restorability. Thus, the measure of over-restorability may provide more information than simply indicating which design has more spare capacity. As expected, the two SLPA designs show a trend towards more over-restorability with increased capacity (over-restorability 129% to 132% with normalized redundancy 71% to 72%). However, ICH uses an entirely different method of positioning capacity which takes more of a global view: The whole capacity set is placed at once. So, in the experiments performed, ICH appears to have an inherently higher over-restorability for a given spare capacity (over-restorability 132% with normalized redundancy 68%) than SLPA. Recall that the ICH design was only 96.6% restorable by SHN, so the over-restorability factor presented in Table 8.5 is even higher when normalized to the restorability obtained by the design. For over-restorability normalized to actual restorability,

the ICH design provides over-restorability of 139%, which is even higher than the SLPA Short over-restorability.

Table 8.5 Over-Restorability Statistics (All Normalized to 1443 Restorable Working Links)

Design Type	Average	Variance	Minimum	Maximum	Median
ICH RPL	132%	4.3%	101%	156%	135%
SLPA Path-Table	130	0.50	117	141	131
SLPA Short	132	0.57	119	143	133

From the large number of alternative SCP's which have similar total spare capacity, the ICH network designs are observed to generally favor those placements which provide very bunched capacity positioning (many spans with zero spare links and many spans with large numbers of spare links). On the other hand, SLPA generally distributes the spare capacity nearly-uniformly amongst the spans. When this investigation started, the author assumed that the even distribution of SLPA would provide a more highly over-restorable design than the bunched distribution of ICH. However, the experimental results show that this is not the case. By some mechanism, which is probably related to the forming of restoration rings of relatively large spare capacity, ICH designs are generally more over-restorable than SLPA designs. It is also observed that the variance of the ICH over-restorability is larger than SLPA's, suggesting that ICH does not consistently achieve this improvement in over-restorability (compared to SLPA). If network designers select the SLPA technique, it is possible to add over-restorability to the objective function for the FS phase, which may allow a final design with higher over-restorability.

8.7 Ease of Implementation

In general, the SLPA methods were significantly more difficult to implement than the ICH methods. The primary reason is that ICH's most essential module, the LP, was available in commercially distributed software package. The other modules of ICH did not significantly impact execution time, so they were implemented in a straight-forward manner. On the other hand, a direct implementation of the modules of SLPA, such as those introduced in the first section of Chapter 7, had execution times which made designs on even the smallest of networks excessively slow. Therefore, each additional execution speed enhancement feature increased the implementation complexity. SLPA Dijkstra implements the path-finding through an easily described (and therefore re-implemented) method which incorporates iteration, binary heaps and localities into a Dijkstra shortest path search. However, the operations required to minimize the number of executions of Dijkstra were both extensive and unique. In addition to these generic complications in the implementation of SLPA, the SLPA Path-table required extensive operations to initialize the path-table, thereby increasing the overall complexity even further.

Table 8.6 presents data of the number of lines of C programming language source code which the implementations presented use. The lines of code are categorized as: cautious implementation and normal implementation. Lines of code generated normally were not essential to the optimization of the performance parameters and were, therefore, implemented in a straightforward, modular manner. The lines of code which were classified as cautiously implemented were generated on a line-by-line basis, with each step requiring a detailed analysis of the complexity involved to maximize the performance of the algorithm. The SLPA Path-table implementation is the most complex by these measures. The SLPA Dijkstra implementation appears to be nearly as complex, but the Dijkstra search itself, which can be primarily acquired from external sources, was categorized as cautious in this analysis. The ICH implementation did not have any cautious implementation modules because of the domination of the LP execution time. The amount of pseudo-code presented in chapters 6 and 7 reflects the number of lines of code which Table 8.6 classifies as cautious.

Table 8.6 Lines of C Programming Language Source Code

Design Type	Cautious Implementation	Total
SLPA Path-Table	1545	3302
SLPA Dijkstra	1387	3215
ICH RPL	0	1717

If network designers choose ICH for network design, customized LP software may improve the design speed and other features. However, such enhancements would negatively affect its ease of implementation.

8.8 Discussion and Recommendations

Because the literature on network restoration does not yet offer a generic list of criteria for rating network designs, users of SCP algorithms may select a technique which best suits the specific needs of their network.

Obtaining a restorable network design is paramount in SCP. Although capacity efficiency is the impetus for the current research, it cannot be favored over restorability. This does not mean that the minimization of redundant capacity is not a worthy objective, but rather that it should be considered only after satisfying a restorable network design.

After securing a network design characterized by restorability and the efficient use of spare capacity, consider the execution time of the algorithm. Execution time may limit the network designer's ability to play with alternative network architectures (span locations, DCS nodes, etc.) before making a provisioning decision. Networks of the future will probably be very adaptable to

the needs of customers, and the capacity requirements of the networks will have to be evolved to a new state relatively quickly (probably daily in the near future).

Ease of implementation is the only remaining comparison category which will probably be of interest to network designers during selection of an algorithm. Until commercial software is available for SCP, program development resources may dictate the algorithm selection.

The major shortcomings of the ICH approach to network design are in the categories that the author ranks highest in importance. ICH occasionally cannot provide a network SCP solution at all, due to unpredictable characteristics of the heuristic and the LP which solves the constraints. Also, the ICH approach has a mismatch between path selection criteria (max-flow) and restoration path selection (k-shortest paths), which results in lower restorability. It also does not illuminate the capacity-restorability trade-off (as SLPA does) which is knowledge of some value to network planners. Based upon these shortcomings, the ICH heuristic is not recommended for general use, although it may find its way into initial investigations because of its ease of implementation.

SLPA Path-table has large space requirements for networks with high node degrees (i.e. greater than four); thus, it too is limited in its ability to provide a design in all situations. Moreover, SLPA Path-table is limited to a single path selection order within the regime of k-shortest paths, dictated by the sorting operations of the path-table. SLPA Path-table has the highest implementation complexity of all of the heuristic algorithms considered here. It is only recommended for low average node degree networks with a large number of nodes, where the execution time of the algorithm is of greatest concern. However, it is easy to imagine that once a route structure is defined for even a very large network, the corresponding path-table can be computed once and distributed on optical disk (if need be) to planners performing detailed capacity placement and growth accommodation calculations.

Overall, the most useful algorithm is SLPA Short. In the crucial area of restorability, this algorithm can alter the path selection order to more closely mimic that of the field deployed restoration algorithm. In fact (as with any SLPA implementation) SLPA Short can even execute (or emulate) the exact restoration algorithm itself for a final check of the design (although execution time does not warrant constant emulation of the restoration algorithm, especially when it is a distributed algorithm). Ironically, the only category of comparison in which SLPA Short does not compare favorably is with respect to capacity efficiency, the original impetus for this research. Even here, however, SLPA Short provides network designs that average only 5.4% more redundant capacity than the lowest redundancy design provided by ICH. Anything in this regime is "near-optimal".

When comparing ICH and SLPA there is an interesting reversal of optimization philosophy that may offer some insight as to why SLPA is well-suited to this problem: ICH seeks to minimize total (integer) cost (sum of spare links) subject to an array of cutset-based constraints to satisfy

restorability. SLPA reverses objective function and constraint: It seeks a steepest ascent in (real-valued) restorability, subject to much more simple constraints on cost addition at each iteration. This leads to an effective and relatively simple polynomial-time heuristic for a problem that is NP-hard in its exact solution.

9 Joint Provisioning and Restorability

9.1 Introduction

When evolving a network design to accommodate new working capacity provisioning, restorability levels must be maintained and redundancy should be minimized. In addition, few spans should be disrupted during the evolution - representing a low cost for installation of new capacity and minimizing network "churn" which entails its own operational risk of errors.

In mesh restoration, the view can be taken that spare capacity for restoration and for growth provisioning exists in a single pool. New working capacity is provisioned from the spare capacity present in this pool. Therefore, each working path provisioned has the potential to decrease restorability by converting a vital spare link to working. The working path routing algorithm in the joint provisioning and restorability experiments (a) adds paths between random nodes; (b) uses shortest path routing; (c) chooses the route which affects restorability the least; and, (d) adds spare capacity immediately to reestablish restorability in the event that the new provisioning event reduced network restorability.

Network designers generally provision working paths one path at a time, even though they add capacity on a modular system basis. In addition, network providers minimize costs by deploying capacity for future needs while upgrading to meet current needs. Therefore, networks generally have redundant capacity beyond the capacity currently required for restoration. That is, they are always "over-provisioned" to some extent with respect to the ideal minimum capacity design. For example, [Luck92] asserts that in 1990 in AT&T's long-haul network, only 54.4% to 56.7% of the fiber deployed was activated. This extra capacity provides the network with super-redundancy (SR), where SR is the amount of spare capacity available beyond that which is required for restoration. In this study, super-redundancy will be calculated for each span i as:

$$SR_{s,i} = \frac{s_i - s_{req,i}}{s_{req,i}} \tag{9.1}$$

where s_i and $s_{req,i}$ are the spare capacity on span i and the required spare capacity for span i in order for the network to be fully restorable.

Another format for expressing the "extra" spare capacity is through the Provisioning Redundancy (PR) measure. In PR, the extra spare capacity on span i is normalized to the working capacity on span i (w_i). In this way, $PR_{s,i}$ provides a measure of how much the network can grow (via working capacity provisioning) without adding capacity, but provides no measure of the restorability of these newly provisioned paths.

$$PR_{s,i} = \frac{s_i - s_{req,i}}{w_i} = \frac{SR_{s,i} \cdot s_{req,i}}{w_i} \tag{9.2}$$

Spans which have large working capacities tend to drive the spare capacity requirements of a network design. Therefore, the spans which have smaller working capacities can often

support larger restorable working capacities without a requirement for extra restoration capacity. The total working capacity which could be restored by the spare capacity present if each span had infinite working capacity is the over-restorability (OR) of the network. $OR_{s,i}$ is quite different from $SR_{s,i}$ and $PR_{s,i}$, because it is a measure of how much restoration benefit to span i that spare capacity on others spans provide, whereas, $SR_{s,i}$ and $PR_{s,i}$ measure the benefit to the other spans of spare capacity on span i. $OR_{s,i}$ is expressed in terms of the number of restoration paths available to span i, k_i.

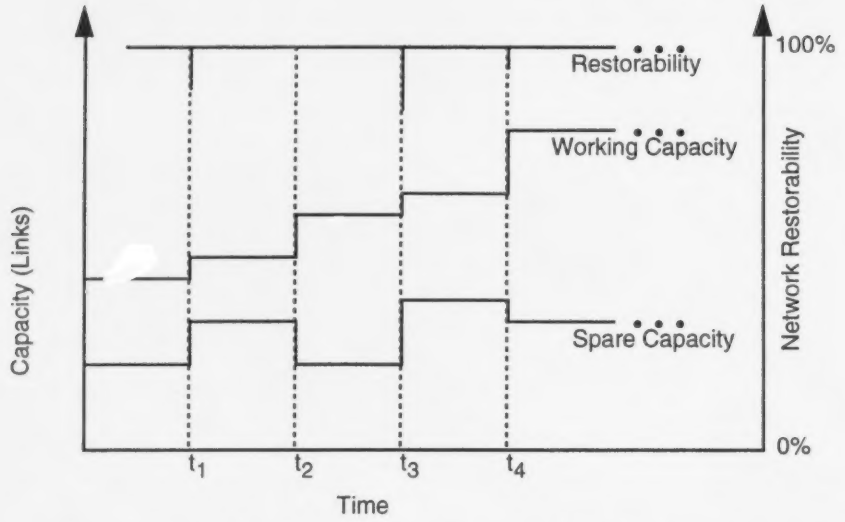
$$OR_{s,i} = \frac{k_i}{w_i} \tag{9.3}$$

9.2 Joint Provisioning and Restorability Strategies

The provisioning of a working path simultaneously reduces the available spare capacity on the spans of the provision (because spare links were converted to working links) and increases the number of required restoration paths for each of these spans (by one path). Thus, each provisioning event can affect the restorability of any network span.

In the experiments reported here, network restorability is always maintained at 100%. Therefore, after the placement for each working path is determined, the restorability of the network is re-established through an SCP algorithm (SLPA Short here). This process is depicted in Figure 9.1. In the figure, the network restorability is always maintained at 100% and therefore each working path addition may trigger addition of spare capacity in order to increase restorability. When the update process places additional capacity into the network, three strategies are investigated here for determining the amount of spare capacity which should be allocated. These are: (1) just-in-time, (2) pre-emptive based on PR, and (3) pre-emptive based on PR and OR.

Just-in-time capacity allocation allows only minimum spare capacity additions corresponding to the minimum capacity which re-establishes the network restorability level. Pre-emptive capacity allocation adds spare capacity where it is required for re-establishing restorability, but it also adds more spare capacity to those spans than what is immediately necessary for restoration. The amount of excess spare capacity is based upon provisioning redundancy, over-restorability or a combination. Although, addition of spare capacity in the following experiments is triggered by a working path provisioning event which decreases the network restorability, a implementation which would be more appropriate for a planning tool would trigger capacity additions before a single provisioning event can reduce network restorability.



Events: At t1, t3, and t4 the network restorability was reduced by the working path addtion and therefore total network capacity was increased.
At t1 and t3 more spare links were added to increase restorability than were converted to working, resulting in a net increase in spare capacity.
At t4 less spare links were added to increase restorability than were converted to working, resulting in a net decrease in spare capacity.
At t2 the working path addition did not impact restorability (spare capacity was converted to working capacity)

Figure 9.1 Capacity Evolution With the Growth of a Network

9.3 Experiments

The joint provisioning and restorability tests here use SLPA Short for the SCP heuristic with incremental migration and system modularities (fiber capacities) of 6. The random network with 50 nodes and nodal degree 4 is the initial network for the experiments. Each experiment accumulates five hundred working path provisioning events between randomly selected nodes. The algorithm maintains 100% network restorability at all times, and capacity is only added to a span where the addition is required to return to 100% restorability.

The just-in-time method for capacity management will use no explicit provisioning redundancy (PR=0.0) or over-restorability (OR=1.5), however some provisioning redundancy will be inherent in the addition of systems (rather than links) when provisioning. Therefore, just-in-time provisioning is accommodated by simply executing the SCP algorithm after each working path provisioning event.

The pre-emptive with PR capacity management strategy uses a 50% provisioning redundancy factor. Therefore, whenever the SCP algorithm determines that a span i requires an

increased capacity in order to re-establish 100% network restorability, the spare capacity on that span (s_i) is changed to:

$$s_i = \max(s_{req,i}, 0.50 \cdot w_i).$$

The pre-emptive with PR and OR capacity management strategy ensures that 150% over-restorability capacity exists in addition to 50% provisioning capacity, whenever a capacity update is triggered for a particular span. Therefore, in this strategy, the spare capacity on a span being updated is:

$$s_i = \max(1.50 \cdot s_{req,i}, 0.50 \cdot w_i).$$

9.4 Network Capacity Changes

The pre-emptive capacity management scheme decreases the number of working path provisioning events which impact restorability. Also, when a working path provisioning event does impact restorability, pre-emptive capacity management reduces the number of spans which spare capacity must be added to in order to re-establish 100% restorability. Table 9.1 shows that pre-emptive capacity management decreases this network "churn" by up to 36% (with PR and OR strategies). The network churn is also decreased by 17% when only the PR strategy for pre-emptive capacity management is used. Only with the PR+OR pre-emptive capacity management strategy does the median working path provisioning event require no spare capacity allocation. With the other capacity management strategies, a median of one span requires spare capacity after each working path provisioning event.

Table 9.1 Number of Spans with Increased Capacity Per Provisioning Event

strategy	PR	OR	mean	variance	minimum	maximum	median
just-in-time	0.0	1.0	0.95	1.2	0	4	1
super-redundancy	0.5	1.0	0.79	0.95	0	6	1
super-redundancy and over-restorability	0.5	1.5	0.61	0.76	0	5	0

With larger values for the PR and OR factors, the amount of network churn can be reduced indefinitely. However, increasing the PR and OR factors generally translates to an increase of network redundancy. Therefore, these parameters must be optimized based upon a cost-benefit analysis of network "churn" and spare capacity requirements.

9.5 Redundancy Levels

Figure 9.2 compares the redundancy requirements of each capacity management strategy. In the figure, redundancy levels are expressed as the total spare capacity in links of the network because working capacities are identical between curves. PR-based pre-emptive capacity management results in minimal increases of network redundancy over just-in-time provisioning. However, PR+OR-based pre-emptive capacity management results in 11% more redundant capacity by the 500-th migration than the other two capacity management methods.

Spare capacity is more efficiently positioned when SCP update capacities are based on provisioning redundancy than over-restorability. Redundancy requirements of the network for future provisioning can be estimated from the current working capacity distribution of the spans because spans are utilized for working capacity based on geographical location when shortest path routing is used. Therefore, spans will generally experience future working capacity growth which is proportional to their current working capacity. On the other hand, redundancy requirements of the network for restorability of future working links (over-restorability) cannot be directly assessed in terms of current capacity levels. This is because the use of capacity for restoration requires that directly adjacent spans also have capacity. Therefore, the process of adding spare capacity may change the routes which are being favored for restoration over time. Also, spans on which spare capacity was added based on increasing the network over-restorability may not require very much spare capacity for working path provisioning purposes because spare capacity requirements of a span are not directly related to working capacity requirements on that same span.

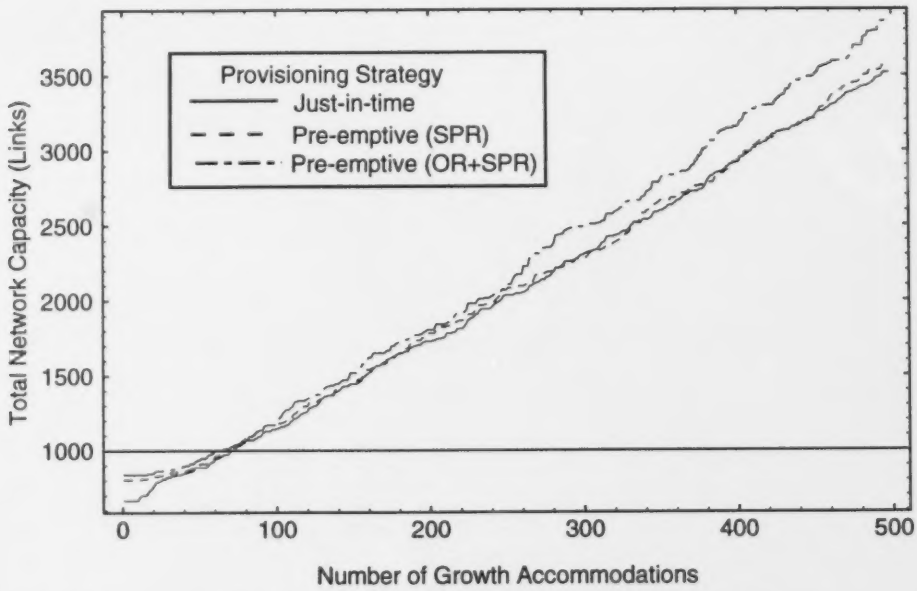


Figure 9.2 Spare Capacity Requirements of a 50-node, davg=4 Network as Working Capacity Requirements Grow

9.6 Summary of Joint Provisioning and Restorability

This section is a preliminary exploration into methods for managing joint provisioning and restorability using an SCP tool to control restorability while meeting growth from the same pool of spare capacity that restoration relies on. It allows pre-emptive capacity management of a single pool of spare capacity which will be used in working path provisioning and restoration. The

provisioning redundancy (PR) and over-restorability (OR) parameters are proposed for control of the trade-off between excess network capacity and the rate at which capacity must be updated (network churn).

PR-based pre-emptive capacity management provided 20% less churn to the network than just-in-time capacity management and requires negligible extra spare capacity (beyond that of just-in-time). PR+OR-based capacity management requires 11% more spare capacity than the other two methods, but achieves 56% less churn than just-in-time capacity management and 30% less churn than PR-based pre-emptive capacity management.

10 Summary

The SCP problem is NP-hard. This was proven by reducing Hamiltonian Cycle existence to a special case of SCP (were each span contains one working link). This justifies the pursuit of SLPA and ICH as heuristics for finding approximate solutions to SCP in reasonable time. An important attribute of an SCP heuristic is the size of network on which it can execute in reasonable time and space.

SLPA Dijkstra has polynomial worst-case complexity with respect to the network size of $O(W^2 \cdot S^5 \cdot N \cdot \log(N))$, where W,S, and N are the number of working links, number of spans and number of nodes respectively. SLPA Short reduces this complexity to $O(W^2 \cdot S^3 \cdot N \cdot \log(N))$ by omitting the most complex capacity redistribution module (add2_sub3()). Both of these SLPA variations obtain $O(W^2 \cdot S)$ when locality information is used in networks with constant nodal degree (d) and restoration path length (RPL). SLPA Path-table has a worst-case time complexity of $O(W \cdot S^7 \cdot d_{max}^{RPL})$, which reduces to $O(W \cdot S^2)$ with locality information and constant d and RPL. The path-table also requires $O(S \cdot RPL \cdot d_{max}^{RPL})$ space in the worst-case. This space requirement increases linearly with network size when using a constant d and RPL. Thus, all variations of the SLPA heuristic are strictly bounded by polynomial time complexity. The SLPA heuristics were shown through experimentation to obtain average-case execution times of $E(N^3)$, which also implies $E(S^3)$ and $E(W^3)$ in these particular experiments. SLPA Short reliably provided the largest number of network designs (24 of 36 study networks) within a limit of 24 hours of SUN SPARC 2™ execution time. All of these networks were designed with RPL = 10 which is considered to exceed RPL requirements in most real networks. At RPL = 6, run times would be significantly less than one day (SLPA complexity is exponential in RPL for values of RPL less than N, that is RPL values which cannot span the network).

An average-case execution of ICH has polynomial execution time also, $E((S \cdot d_{avg})^2)$, but its worst-case time complexity is exponential, $O(2^N)$. Indeed, there were two (of 36) test cases in which ICH obtained no result in over four days of execution. Otherwise, ICH completed 16 of 36 study network designs within the 24 hour limit.

SLPA network designs are at least 99.7% restorable by arbitrary order k-shortest paths algorithms. The discrepancy arises when path selection order between identical length restoration paths can impact restorability. Thus, only restoration algorithms which exactly match the selection criterion used in network design are guaranteed to obtain full restoration. SLPA's maximal compatibility with k-shortest path restoration is consistent with real-time distributed path finding techniques to date such as [Grov87, YaHa88, Grov89, GrVe90, SaNi90]. SLPA is architecturally capable of incorporating the actual field-deployed restoration algorithm itself as a final design stage to ensure full restoration.

ICH network designs are strictly restorable only by max-flow routing with no limit on maximum path length. Thus, observed restorability of ICH designs by k-shortest paths algorithms was 94%, or as much as 96.6% when using SHN™, a parallel distributed restoration algorithm for field use (as opposed to planning studies). Because ICH designs for maximum flow (rather than k-shortest paths) and LP solutions are strict-sense optimal when obtained from an adequate constraint set, ICH network designs require slightly less redundant capacity than SLPA designs. On average, SLPA placed 5% more capacity than ICH RPL. SLPA Short required a further 2% of excess capacity, as the price for elimination of the most complex design tightening redistribution stage of SLPA (add2_sub3()).

SLPA accommodated network growth with the least overall amount of network "churn," accommodating an average of 3.6 additional working links with addition of capacity to 4.6 spans. ICH placed an average of 5.3 additional links to provision an average of 3.8 fully restorable working links.

Because SLPA is a new method for which new problem specific optimization methods were invented, the implementation involved approximately 100% more lines of source code than ICH. Half of the code for SLPA had to be optimized for execution speed and, therefore, it required cautious implementation. This translates to much longer software development time. ICH is an LP optimization problem which primarily relies on commercially available software for the routines that are crucial to time-complexity.

Chapter 9 demonstrated joint provisioning and restorability using SLPA for SCP updates. Just-in-time capacity management ensures a minimum of network capacity at all times, but in tests it required capacity additions to 0.95 spans in each working path provisioning event. Pre-emptive capacity management based on super-provisioning redundancy reduced these capacity additions to 0.79 spans for each working path provisioned. Pre-emptive capacity management achieves this 30% reduction in network "churn" with negligible extra capacity. In principle, network designers can use the control mechanisms introduced here to achieve the minimum cost for capacity management based upon a balance of the trade-off between network "churn" and redundant capacity. More work is required on these concepts.

In closing, ICH and SLPA have differing merits. ICH is a classically-inspired formulation of SCP as an LP optimization problem. In this regard it has the appealing prospect of strictly optimal solutions (aside from final rounding effects) when provided with the full constraint set. On the other hand, due to its reliance on LP, ICH can and does exhibit exponential execution times unpredictably for some data sets. By comparison, SLPA is more specific to the particular problem. It is based on a greedy synthesis principle that a near-optimal global design might result from iterative placement of each additional spare link so as to maximize the (local) increase in overall network restorability. When this principle is enhanced by subsequent design tightening,

the algorithm has several practical properties: SLPA's worst-case complexity is polynomial-time. Even with a worst-case RPL=10, SLPA completed execution for all test cases and showed run times that increased smoothly and predictably with network size. SLPA's outputs were all within 93% of the theoretical bound for capacity efficiency at full restorability, while designing networks specifically for accommodation of k-shortest path restoration.

10.1 Further Research

10.1.1 Max-flow Strictly Constrained by RPL

An algorithm or heuristic which can evaluate max-flow which is constrained by an RPL limit would complement the current research. [DuGr91] finds that k-shortest paths flow is typically more than 98% of max-flow over a wide range of quasi-planar network models. However, in restoration, it is desirable to limit the RPL. k-shortest paths algorithms accommodate RPL limits inherently, cutting off available restoration when the k+1st path is longer than RPL links. There is currently no equivalent method of limiting the path lengths in a max-flow design and, therefore, the observed difference between max-flow and k-shortest paths was 5.5% (Section 6.6) when RPL limits were included. This result includes locality restrictions which is the approximate method used here to limit RPL in max-flow situations.

10.1.2 SCP Synthesis Algorithm With Simulated Annealing

Simulated annealing might provide an alternate synthesis tool for SCP in network design. The following excerpt from [AaKo89] describes simulated annealing.

> "Annealing is the physical process of heating up a solid until it melts, followed by cooling it down until it crystallizes into a state with a perfect lattice. During this process, the free energy of the solid is minimized. Practice shows that the cooling must be done carefully in order not to get trapped in locally optimal lattice structures with crystal imperfections.
>
> In combinatorial optimization, we can define a similar process. This process can be formulated as the problem of finding - among a potentially very large number of solutions - a solution with minimum cost. Now, by establishing a correspondence between the cost function and the free energy, and between the solutions and the physical states, we can introduce a solution method in the field of combinatorial optimization based on a simulation of the physical annealing process. The resulting method is called Simulated Annealing."

If there are networks where SLPA finds a local minimum SCP rather than the absolute minimum SCP, Simulated Annealing might be able to avoid the local minimum. Thus, simulated annealing might be used alone or as an addition to the SLPA program designed to avoid local

minima. The manner in which Simulated Annealing would be used with SLPA is to allow redistributions of spare capacity which do not strictly adhere to the restorability and redundancy objectives. Then, as the simulation continues, the cooling rate must be adjusted so that fewer and fewer of these illegal operations are allowed. Eventually, no illegal operations are allowed and if the cooling is slow enough, an absolute minimum SCP will be found.

10.1.3 SLPA with Span Specific RPL

The implementation of SLPA presented here only allows a global RPL. In order to allow span specific RPL's, the local node and local span lists must be separated into two lists to reflect the two roles of these lists in the current implementation: (a) Local spans to Span A are the spans which can assist in restoring Span A. And, (b) local spans to Span A are the spans to which a change in spare capacity on Span A can alter their restorability. Each use of locSpan and locNode in the SLPA program must be checked for the orientation required.

10.1.4 Usable Algorithm for Polynomial Time-Complexity Linear Program

One of the major restrictions of the ICH method for SCP is the unpredictability of run-times which result from the exponential (worst-case) time-complexity of Simplex LP's. The ICH method would benefit greatly from a polynomial implementation of an LP which has assumed reasonable average time-complexity.

10.1.5 Exact Locality Identification in Polynomial Time

The SLPA Path-table algorithm has correct span and node locality information. However, Section 7.2.4.4 showed that this method of generating locality information has time complexity of $O(d^{RPL})$. The SLPA Dijkstra algorithm provided approximate span and node locality information by triangulating the distance to each node from the two end-nodes of a span. If the combined distances from the end-nodes is less than RPL, the node is a local node. This method may provide erroneous locality information when the triangulation does not use span disjoint routes for the two end-nodes. This situation was shown in Figure 7.13.

The SLPA Dijkstra may be improved by an accurate locality identification technique. This new technique should execute strictly in polynomial time.

10.1.6 Network Topology Design in Which k-shortest Path Flow Equals Max-flow

Some network topologies, called matroids, have equivalent k-shortest paths flow and max-flow. If these topologies can be applied to telecommunications networks, an SCP would be guaranteed to be restorable by a k-shortest path algorithm when designed by ICH or SLPA.

10.1.7 Fast Simulation of Known Restoration Algorithms for Use in SLPA for Checking Restorability

Chapter 7 mentioned that SLPA can accommodate a final check on restorability which executes the specific algorithm which will be used in the network. This algorithm can be incorporated into a final FS phase, ensuring 100% restorability of the network. This approach avoids the details of k-shortest path dependence on path selection order by making sure the order is the same in network design and network restoration. In order to make this strategy feasible, the restoration algorithm must execute quickly. Therefore, simulations of known restoration algorithms must be generated which have high compatibility to the algorithm while obtaining fast execution times. For instance, the SHN™emulation, which was used for restorability evaluation in this thesis, incorporates asynchronous parallel-execution timing details. These emulation details are necessary and valuable for protocol tests, but make incorporation into SLPA unrealistic due to execution time restrictions when using a single processor machine. However, parallel computers could conceivably execute the SHN in its truly parallel orientation for R_n evaluation faster than conventional algorithms such as our metaDijkstra routine . Research into the feasibility of parallelizing metaDijkstra would be required to decide if the SHN protocol itself is a faster kernel for calc_rest() on a parallel machine than metaDijkstra is on a single machine (or a parallel machine).

Bibliography

[Agar89] Agarwal, Y.K.; "An Algorithm for Designing Survivable Networks"; AT&T Technical Journal; May/June 1989; pp 64-76.

[AaKo89] Aarts, E.H.L. and Korst, J.; "Simulated Annealing and Boltzmann Machines"; John Wiley & Sons Ltd.; 1989.

[AyFr70] Ayoub, J.N. and Frisch, I.T.; "Optimally Invulnerable Directed Communications Networks"; IEEE Transactions on Communication Technology; Vol. COM-18, No. 5, October 1970; pp.484-489.

[Birc88] Birch, D.; "The King's Chessboard"; NAL Penguin Inc.; 1988.

[BoTh70] Boesch, F.T. and Thomas, R.E.; "On Graphs of Invulnerable Communication Nets"; IEEE Transactions on Circuit Theory; Vol. CT-17, No. 2, May 1970; pp. 183-192.

[BrBr88] Brassard, G. and Bratley, P.; "Algorithmics: Theory and Practice"; Prentice-Hall Inc.; 1988.

[Brua77] Brualdi, R.A.; "Introductory Combinatorics"; Elsevier Science Publishing Co., Inc.; 1977.

[CaMo89] Cardwell, R.H., Monma, C.L. and Wu, T-H.; "Computer-Aided Design Procedures for Survivable Fiber Optic Networks"; IEEE Journal on Selected Areas of Communications, Vol. 7, No. 8, October 1989.

[Chen90] Chen, W-K.; "Theory of Nets: Flows in Networks"; John Wiley & Sons; 1990.

[ChFr70] Chou, W. and Frank, H.; "Survivable Communication Networks and the Terminal Capacity Matrix"; IEEE Transactions on Circuit Theory; Vol. CT-17, No. 2, May 1970; pp. 192-197.

[Cook71] Cook, S.A.; "The complexity of theorem-proving procedures"; Proceedings of 3rd Annual ACM Symposium on the Theory of Computing, pp. 151-158; 1971.

[DaLo86] Day, M.H. and Lockhart, C.M.; "Survivable Network Planning at AT&T Bell Laboratories"; 1986 IEEE Military Communications Conference; pp 24.1/1-5, vol. 2.

[DuGr91] Dunn, D.A., Grover, W.D., and MacGregor, M.H.; "Comparison of k-shortest Paths and Maximum Flow Routing for Network Facility Restoration"; submitted to IEEE J-SAC Special Issue: Integrity of Public Telecommunications Networks; June 18, 1992. TRLabs internal report, 1991.

[Flan90] Flanagan, T.; "Planning a SONET Network"; IEEE International Conference on Communications 1990; pp. 909-914.

[FlOx89] Flanagan, T., Oxner, S., and Elkaim, D.; "Principles and Technologies for Planning Survivability - A Metropolitan Case Study"; Proc. of IEEE GLOBECOM 1989; pp. 813-820.

[FoFu56] Ford, L.R.Jr. and Fulkerson, D.R.; "Maximal flow through a network"; Canad. Journal of Mathematics; 1956.
Ford, L.R.Jr. and Fulkerson, D.R.; "Flows in Networks"; Princeton University Press; 1962.

[Fran70] Frank, H.; "Some New Results in the Design of Survivable Networks"; 12th Midwest Symposium on Circuit Theory; I.3.1-8.

[FrCh70] Frank, H. and Chou, W.; "Connectivity Considerations in the Design of Survivable Networks"; IEEE Transactions on Circuit Theory; Vol. CT-17, No. 4, November 1970; pp. 486-490.

[FrFr70] Frank, H. and Frisch, I.T.; "Analysis and Design of Survivable Networks"; IEEE Transactions on Communication Technology; Vol. COM-18, No. 5, October 1970; pp. 501-519.

[Gibb85] Gibbons, A.; "Algorithmic Graph Theory"; Cambridge University Press; 1985.

[GoMi84] Gondran, M. and Minoux, M.; "Graphs and Algorithms"; John Wiley & Sons; 1984.

[GrBi90] Grover, W.D., Bilodeau, T., and Venables, B.D.; "Near Optimal Synthesis of a Mesh Restorable Network"; Telecom Canada Report CR-90-16-01.

[GrBi91] Grover, W.D., Bilodeau, T., and Venables, B.D.; "Near Optimal Spare Link Placement in a Mesh Restorable Network"; Proc. of IEEE GLOBECOM 1991.

[Grov87] Grover, W.D.; "The Selfhealing Network: A fast distributed restoration technique for networks using digital cross-connect machines"; Proc. of IEEE GLOBECOM 1987; pp. 1090-1095.

[Grov89] Grover, W.D.; "Selfhealing Networks - A Distributed Algorithm for k-shortest Link-Disjoint Paths in a Multi-Graph with Applications in Realtime Network Restoration"; Ph.D. Dissertation; University of Alberta; Fall 1989.

[Grov89b] Grover, W.D.; "Influence of Selfhealing & Scavenging Technology on Network Availability Planning"; Telecom Canada Report CR 89-16-04; February, 1990.

[GrVe90] Grover, W.D., Venables, B.D., Sandham, J.H., and Milne, A.F.; "Performance studies of a Selfhealing network protocol in Telecom Canada long haul networks", Proc. of IEEE GLOBECOM 1990.

[Karm84] Karmarkar, N.; "A New Polynomial-Time Algorithm for Linear Programming"; Combinatorica; 4, 373-395; 1984.

[Kess88] Kessler Marketing Intelligence, "Fiber Optic Long Haul Systems Planned and in Place"; Newport, Rhode Island, USA, issue 2, 1988.

[Khac79] Khachiyan, L.G.; "A Polynomial Algorithm in Linear Programming"; Soviet Mathematics Doklady; 20, 191-194; 1979.

[Krte88] Krten, O.J.; "Hypothetical Reference Digital Path for Dynamic Network Architecture - Availability Considerations -"; Telecom Canada/BNR Report OCTL87-0002 P 87-02 B; February 1988.

[Luck92] Lucky, R.W.; "Technical Presentation on Fiber Optic Transmission and Switching Systems Including: Optical Fibers, Laser Transmitters, Receivers, Optical Amplifiers, SONET, and ATM"; IEEE Videoconferences (VC0007-5): Making Optoelectronic High Performance Data Communications Work; March 26, 1992.

[MaGr89] MacGregor, M.H., Grover, W.D., and Sandham, J.H.; "Capacity Scavenging in the Telecom Canada Network; A Preliminary Assessment"; Telecom Canada Report CR 89-16-03; November, 1989.

[MaWi91] MacGregor Smith, J. and Winter, P. (eds); "Topological Network Design"; Annals of Operations Research; 33, 1-4; December, 1991

[McGo88] McGorman, R.E.; "A Methodology for Designing Survivable Telephone Networks"; IEEE International Conference on Communications 1988; pp. 1172-1176, vol.2.

[NeSc87] Newport, K.T. and Schroeder, M.A.; "Network Survivability Through Connectivity Optimization"; Proceedings - IEEE International Conference on Communications 1987; pp 471-477.

[NeVa89] Newport, K.T. and Varshney, P.K.; "On the Design of Performance-Constrained Survivable Networks"; 1989 IEEE Military Communications Conference; pp 663-670, vol.3.

[Proa89] Proakis, J.G.; "Digital Communications, 2nd Edition"; McGraw-Hill Inc.; 1989.

[Ravi92] Ravi, N.; "Telecommunications in India"; IEEE Communications Magazine; March, 1992; pp 24-29.

[RoAb86] Roohy-Laleh, E., Abdou, E., Hopkins, J. and Wagner, M.A.; "A Procedure for Designing a Low Connected Survivable Fiber Network"; IEEE Journal on Selected Areas in Communications, Vol. SAC-4, No. 7, October, 1986; pp 1112-1117.

[RoRo86] Roohy-Laleh, E. and Ross, N.; "Network Design for Survivability: Procedure and Case Study in a Dynamic Network Architecture"; IEEE International Conference on Communications 1986; pp. 923-930, vol. 2.

[SaNi90] Sakauchi, H., Nishimura, Y. and Hasegawa, S.; "A Self-Healing Network with An Economical Spare-Channel Assignment"; Proc. of IEEE GLOBECOM 1990; pp 438-443.

[SaVe89] Sandham, J.H., Venables, B., and Grover, W.D.; "Selfhealing Technology in the Telecom Canada Intertoll Network"; Telecom Canada Report CR-89-16-01 (released publicly as TM-91.015); October 1989.

[Schr86] Schrijver, A.; "Theory of Linear and Integer Programming"; John Wiley & Sons Ltd.; 1986.

[StWe69] Steiglitz, K., Weiner, P. and Kleitman, D.J.; "The Design of Minimum-Cost Survivable Networks"; IEEE Transactions on Circuit Theory; Vol. CT-16, No. 4, November 1969

[Talb89] Talbott, R.R.; "Network Survivability Analysis"; Fiber and Integrated Optics, Volume 8, pp. 13-48

[TsCo90] Tsai, E.I., Coan, B.A., Kerner, M. and Vecchi, M.P.; "A Comparison of Strategies for Survivable Network Design: Reconfigurable and Conventional Approaches"; Proc. of IEEE GLOBECOM 1990; pp 49-55.

[Vogt91] Vogt, N.; "Introduction of SDH in Germany"; Digital Cross-connect Workshop IV; Banff, Alberta; June 17-20, 1991.

[WaNa88] Watanabe, T., Nakamura, A. and Emura, H.; "On the Diameters of Survivable Networks"; 1988 IEEE International Symposium on Circuits and Systems, Proceedings (ISCAS '88); pp 171-174.

[Wrob90] Wrobel, L.A.; "Disaster Recovery Planning for Telecommunications"; Artech House, Norwood, MA, USA; 1990.

[WuHa89] Wu, T-H. and Habiby, S.F.; "Strategies and Technologies for Planning a Cost-Effective Survivable Fiber Network Architecture Using Optical Switches"; IEEE International Conference on Communications 1989; pp. 749-755.

[YaHa88] Yang, C.H. and Hasegawa, S.; "FITNESS: Failure Immunization Technology for Network Service Survivability";. Proc. of IEEE GLOBECOM 1988.

Appendix A
Known Telecommunications Network Topologies

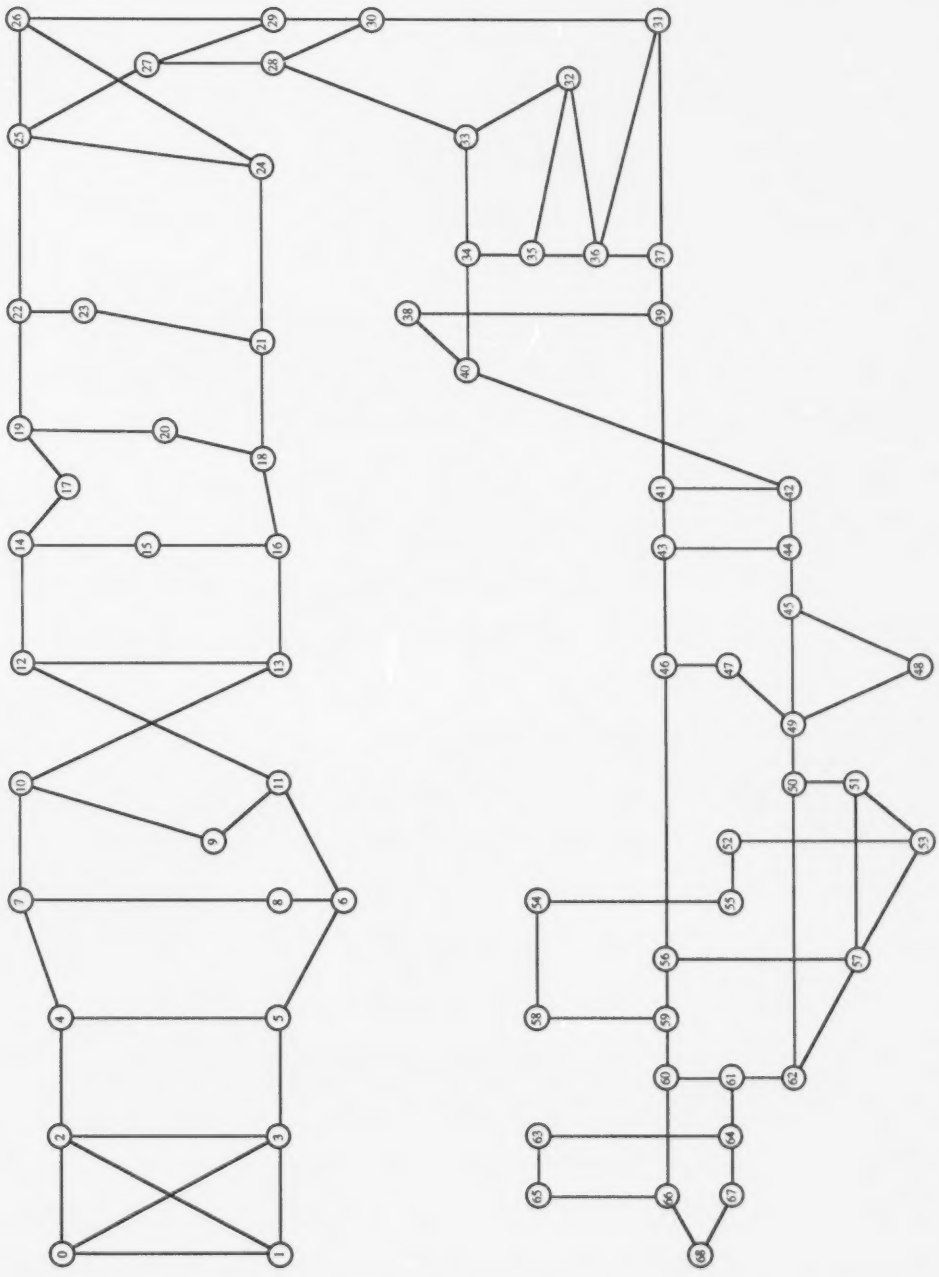


Figure A.1 The Telecom Canada Network [SaVe89]

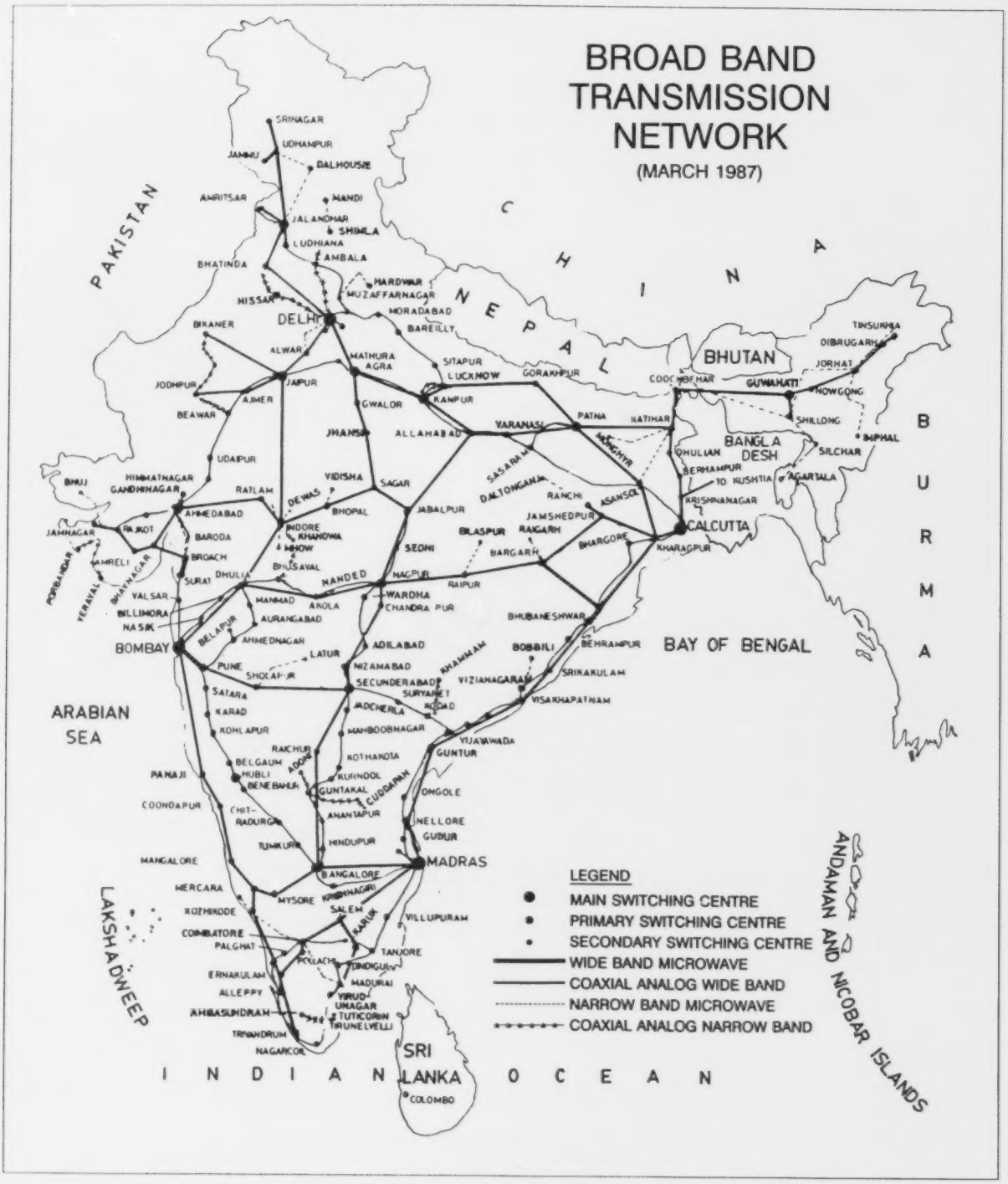


Figure A.2 The Indian Network [Ravi92]

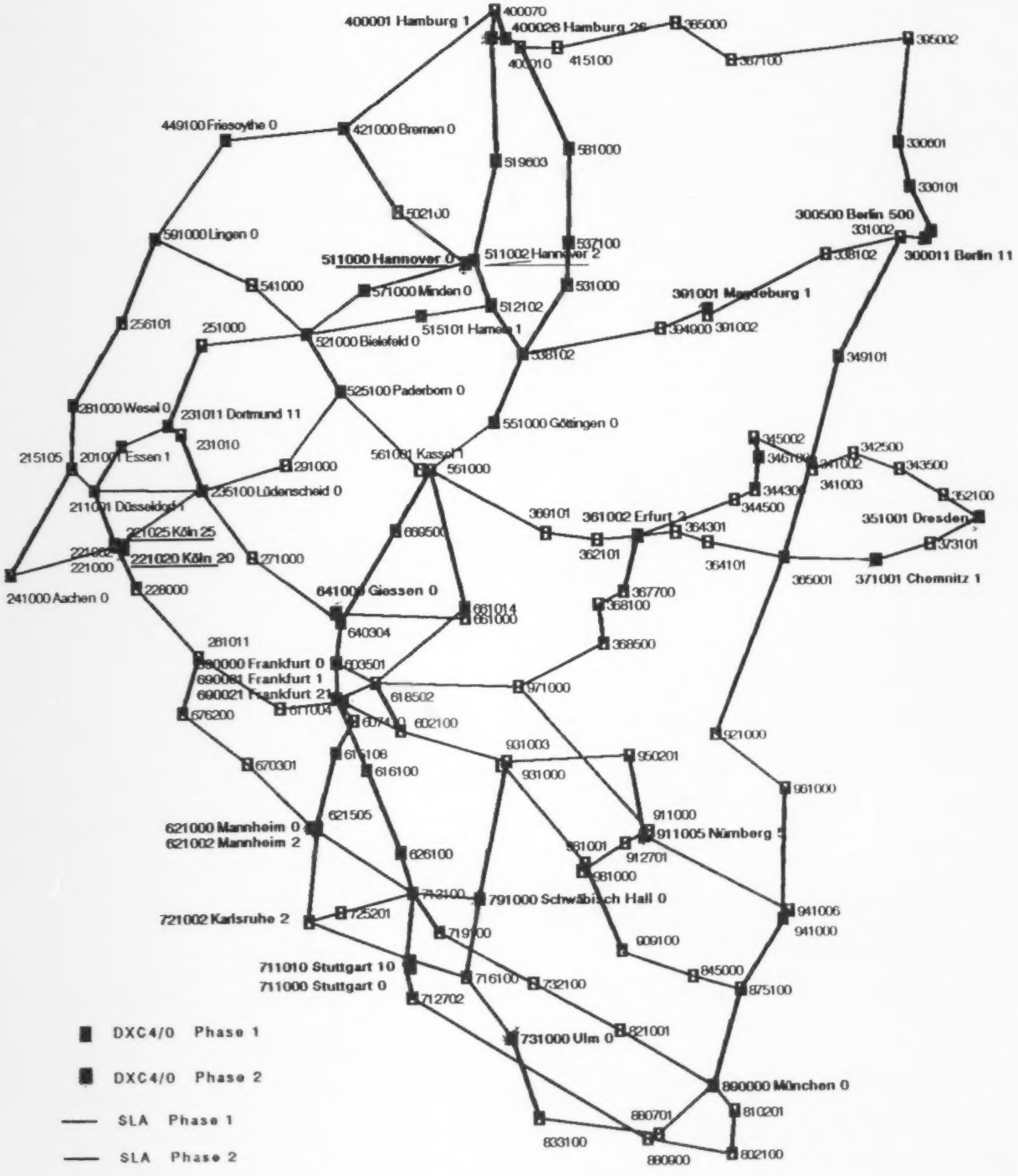


Figure A.3 The German Network [Vogt91]

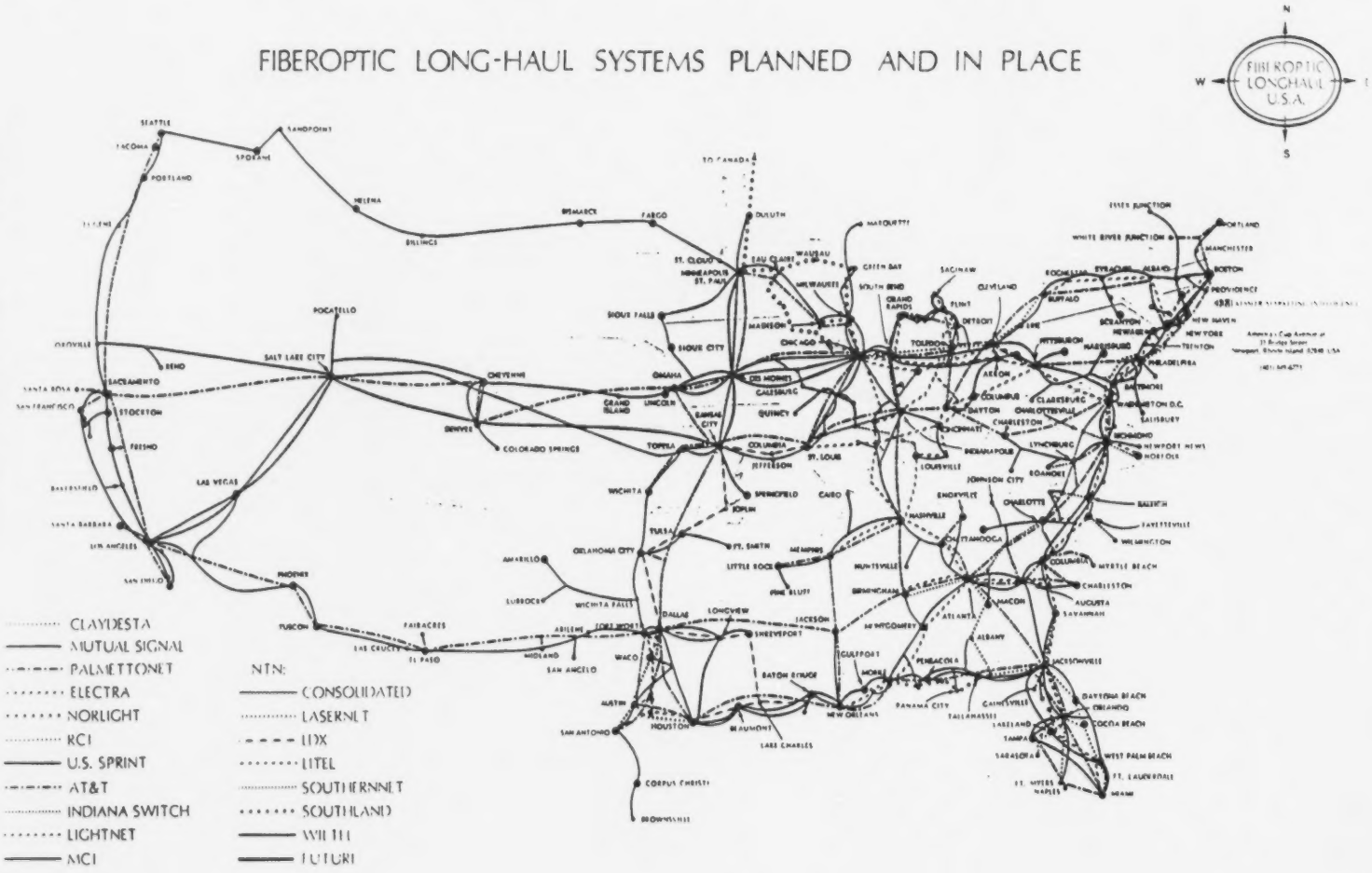


Figure A.4 The U.S. Network [Kess87]

Appendix B
Experimental Telecommunications Network Topologies

Each of the figures in this appendix contains the four network topologies of equivalent total number of nodes and varying average node degree in the following format:

$d_{avg}=3$	$d_{avg}=4$
$d_{avg}=5$	$d_{avg}=6$

The networks presented here are as generated by NGA before the pruning of unrestorable spans and the corresponding nodes.

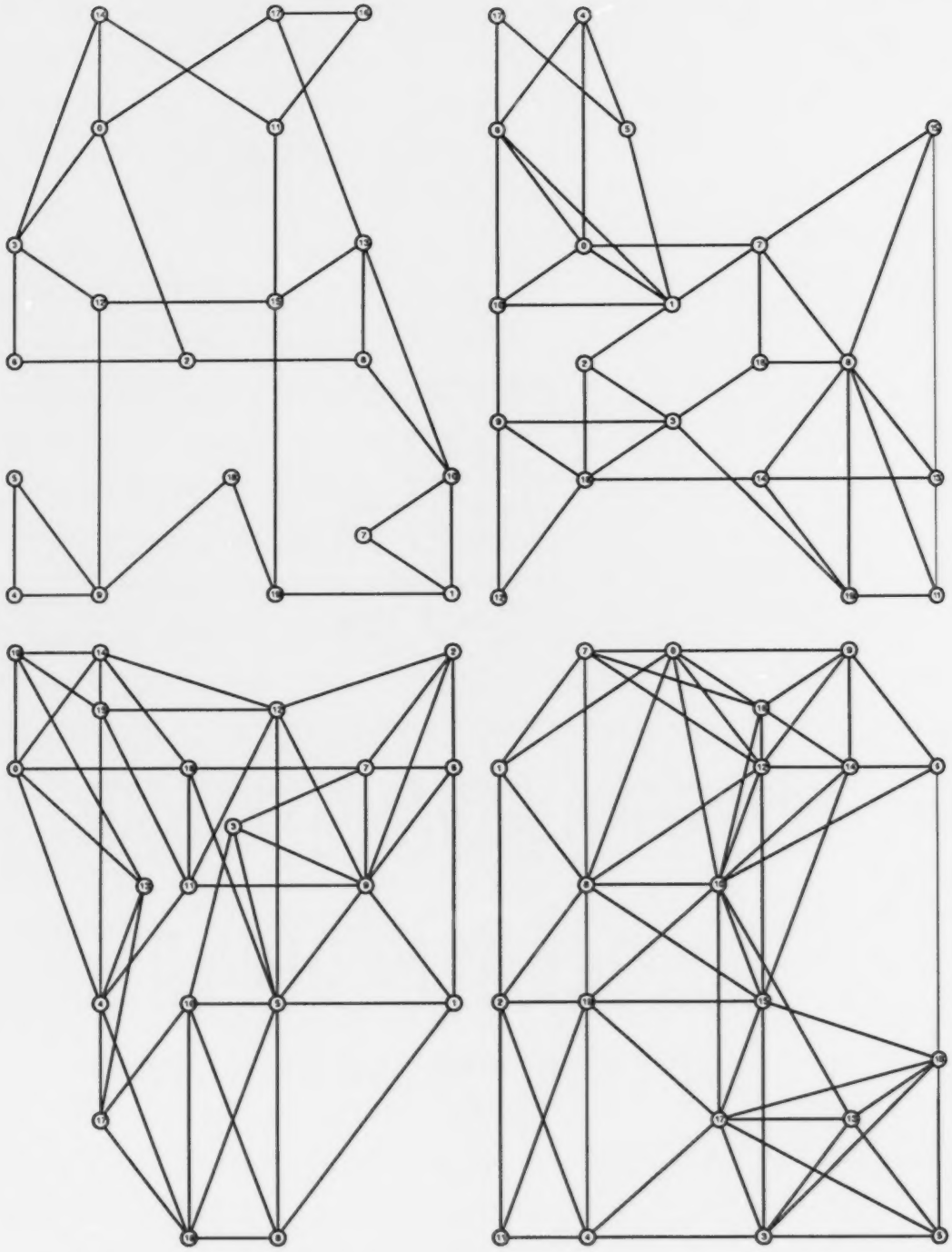


Figure B.1 20-Node Networks

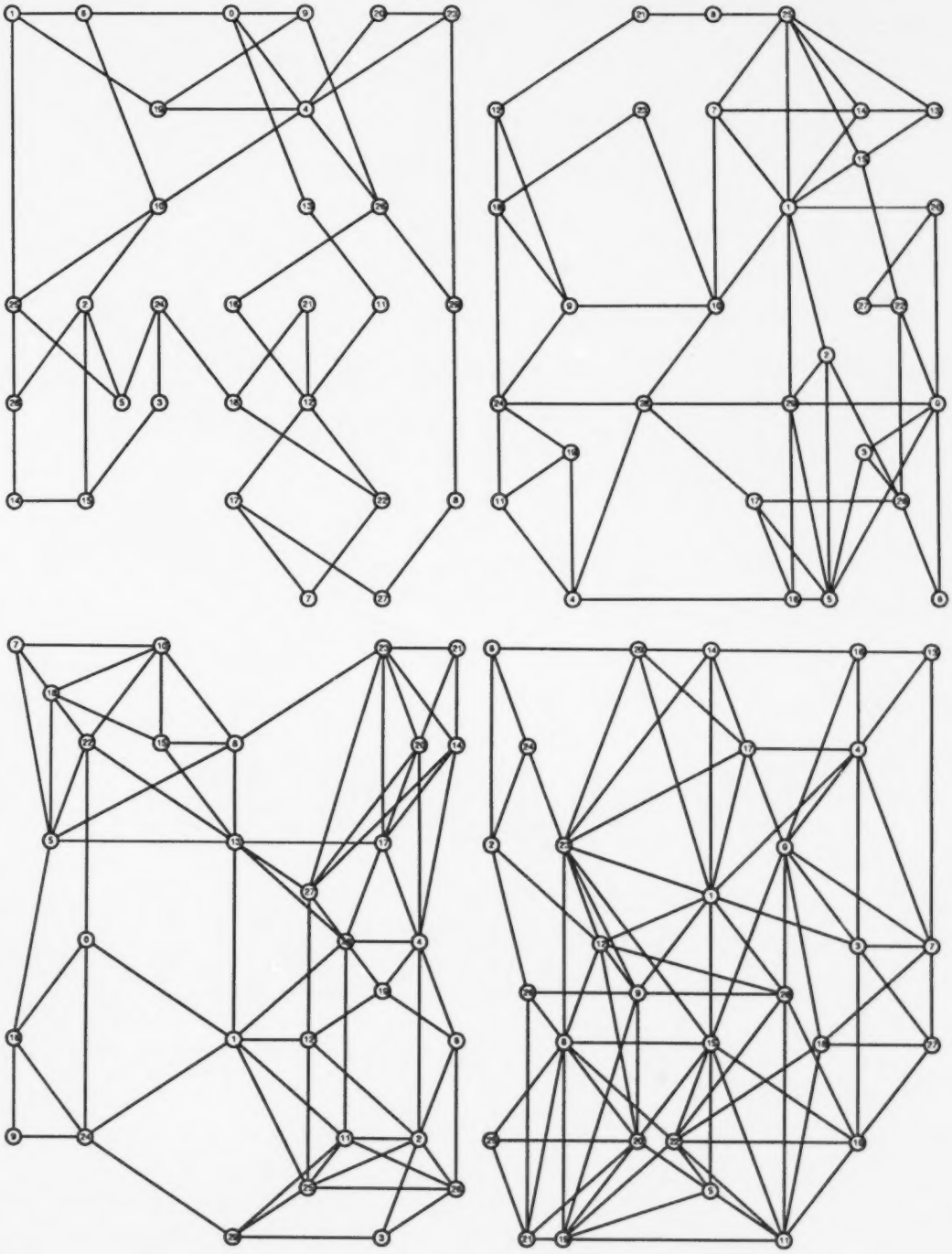


Figure B.2 30-Node Networks

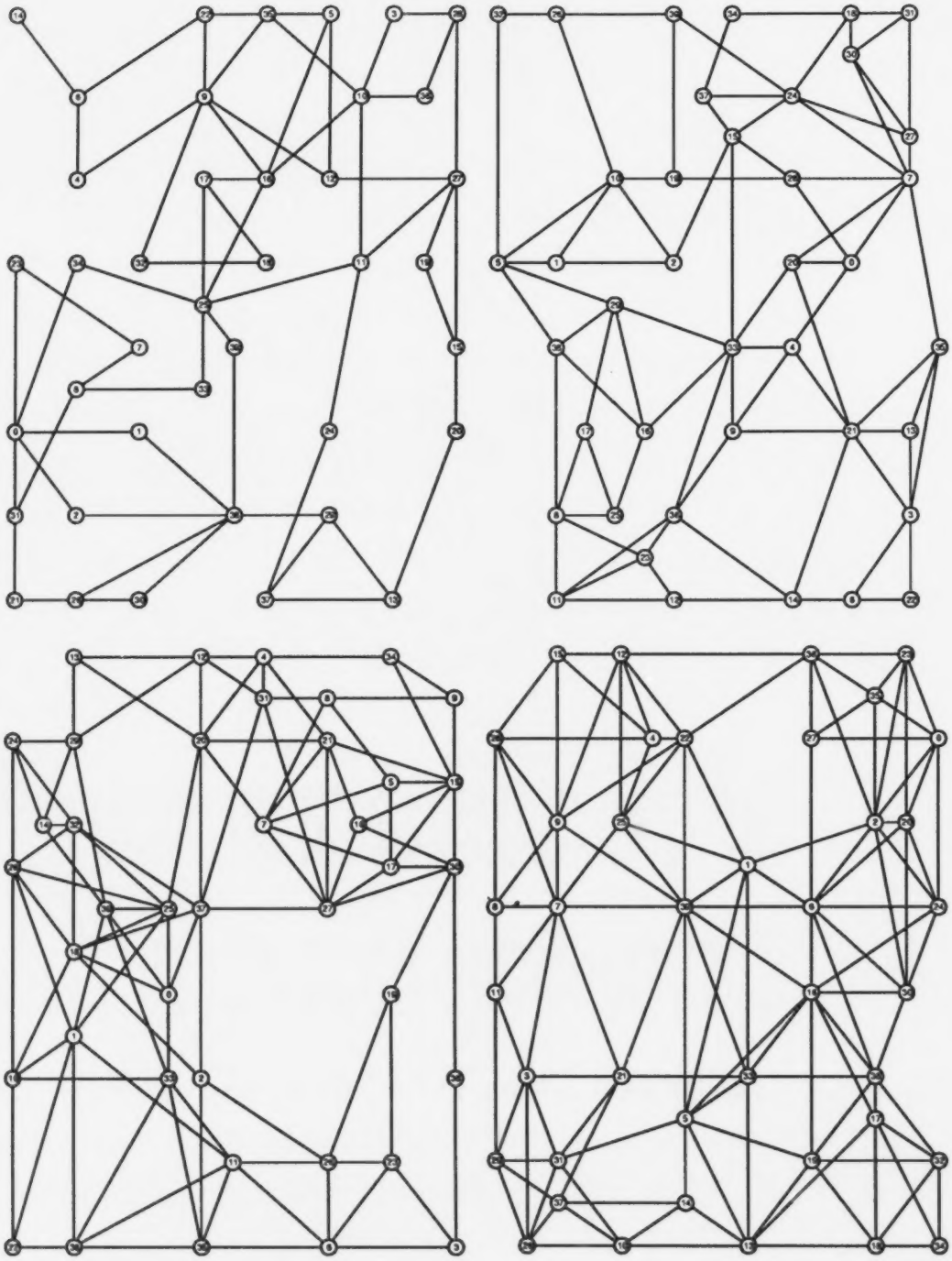


Figure B.3 40-Node Networks

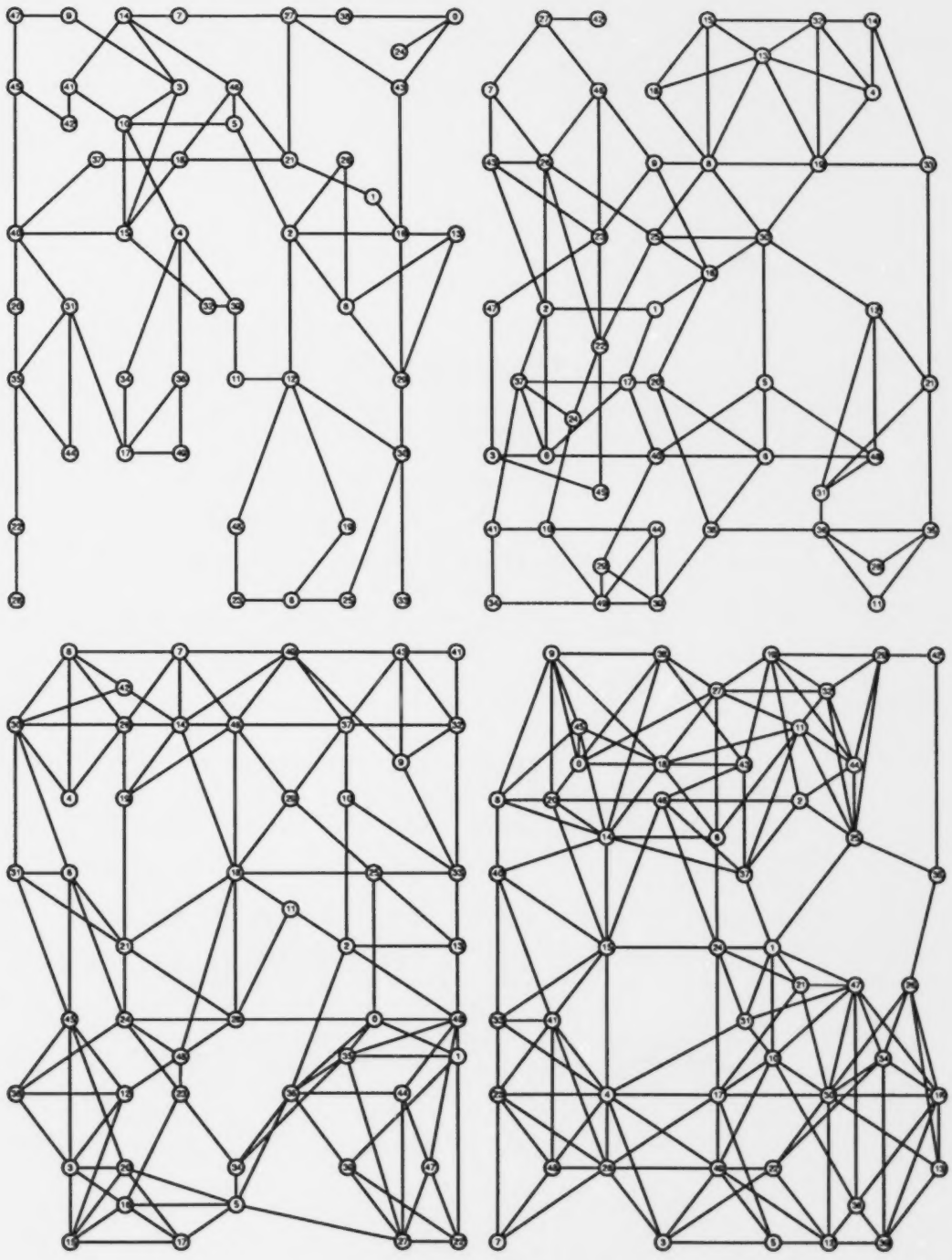


Figure B.4 50-Node Networks

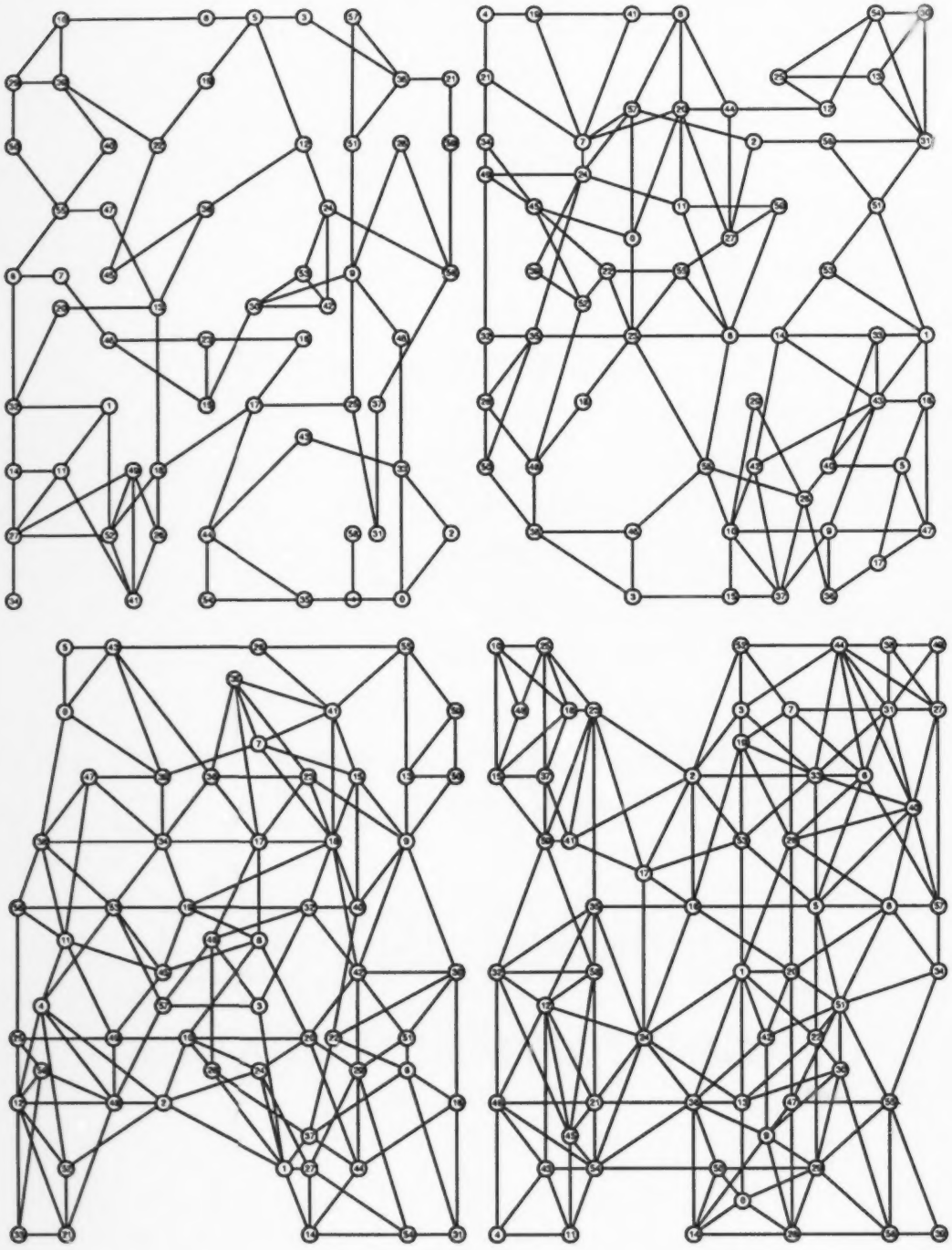


Figure B.5 60-Node Networks

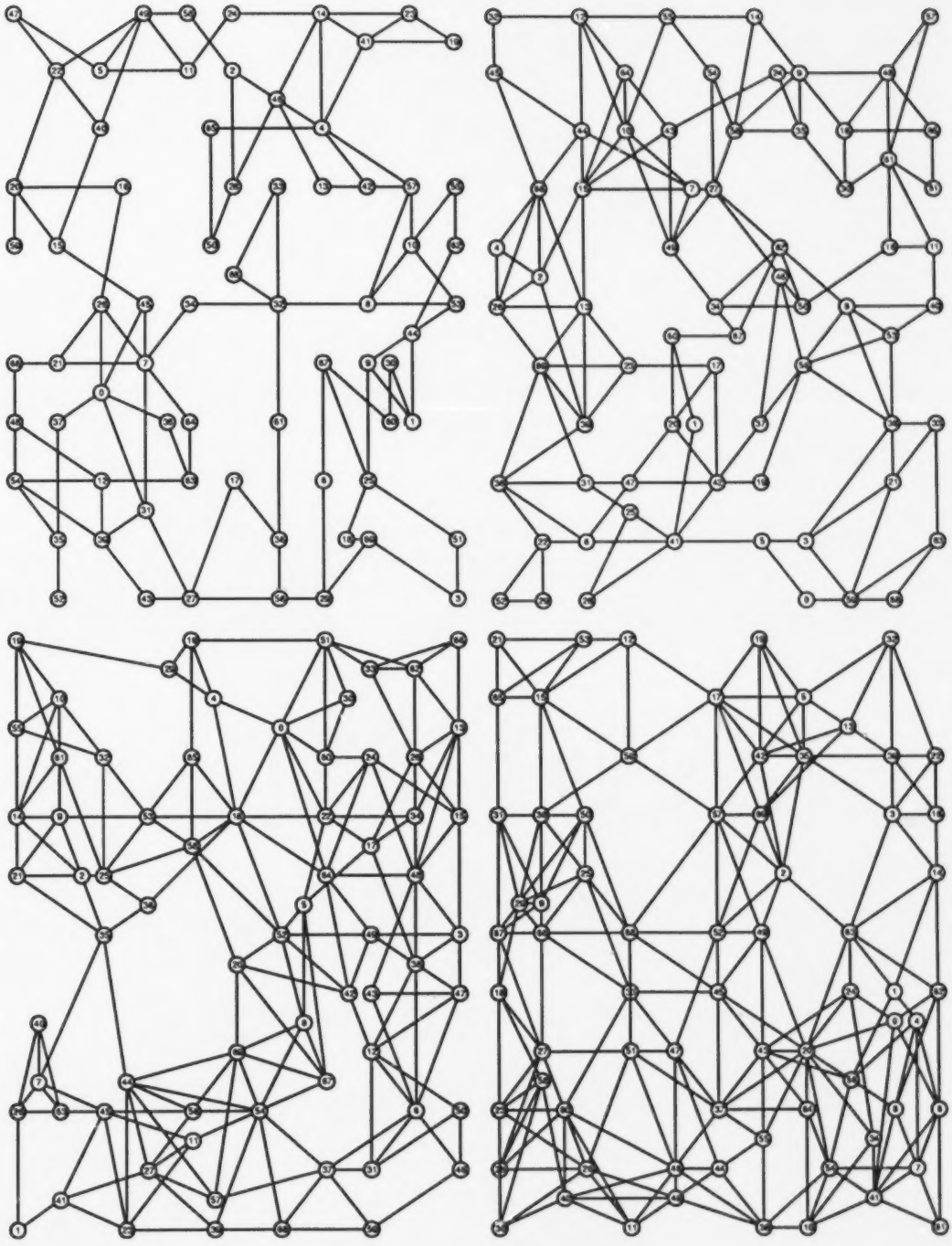


Figure B.6 70-Node Networks

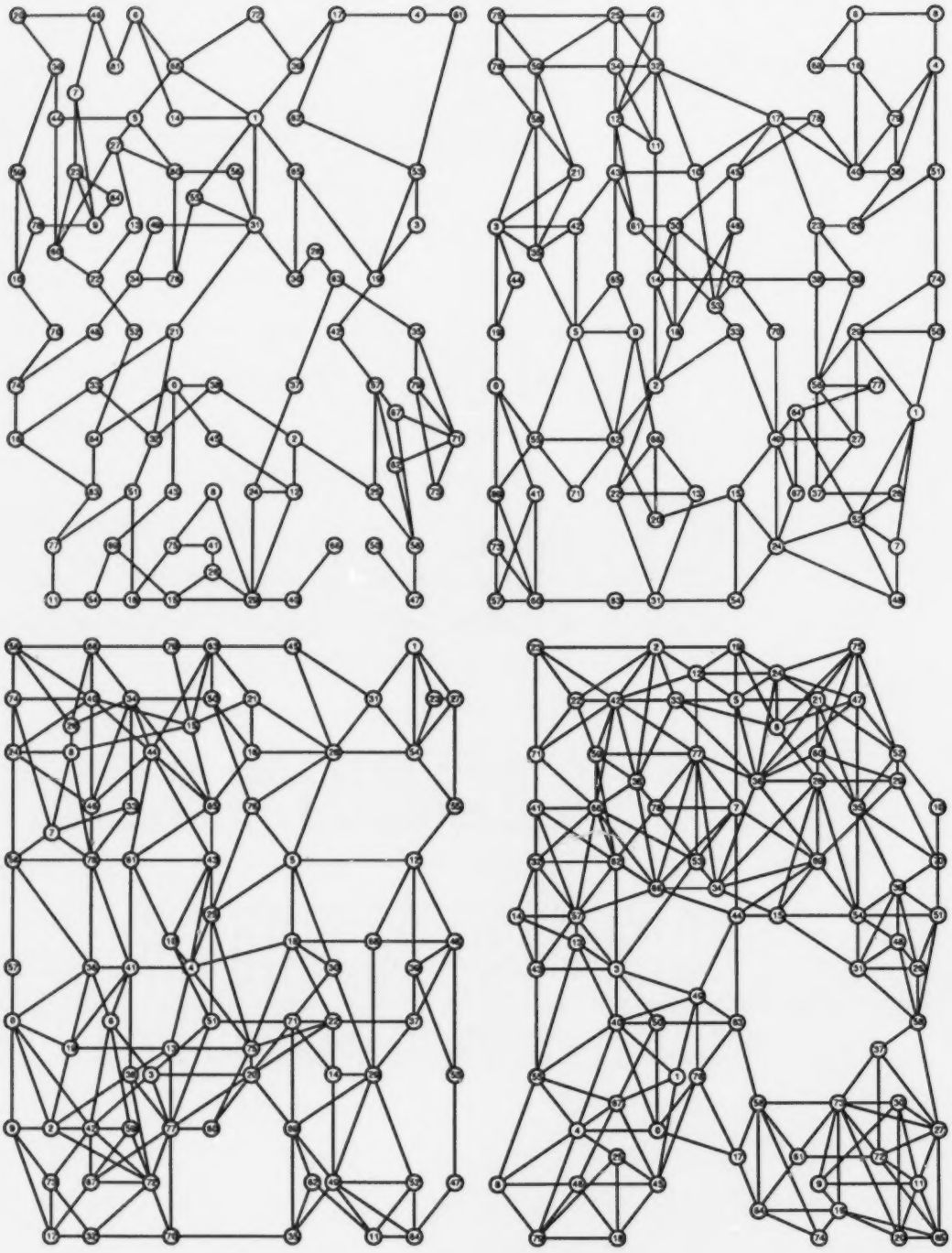


Figure B.7 80-Node Networks

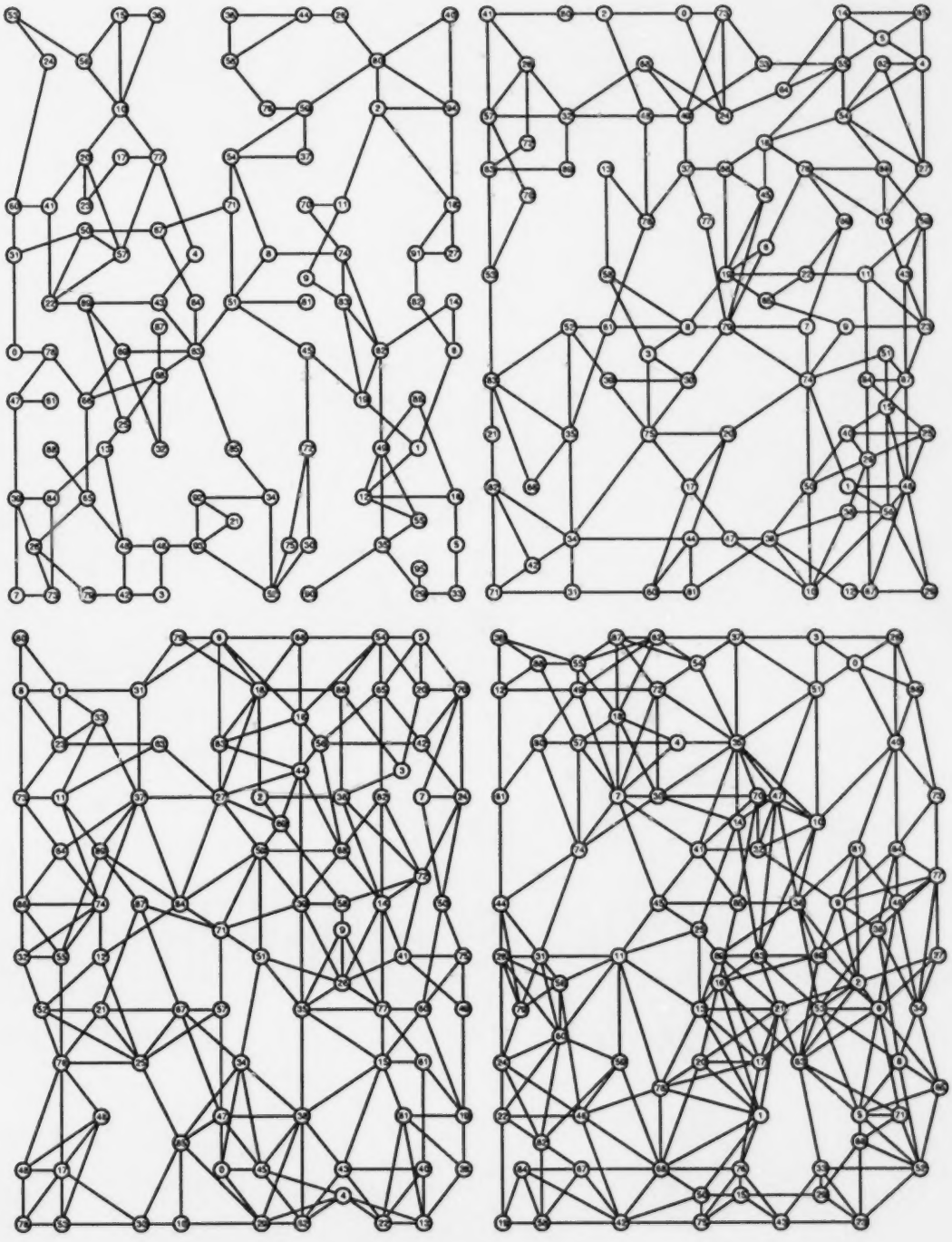


Figure B.8 90-Node Networks

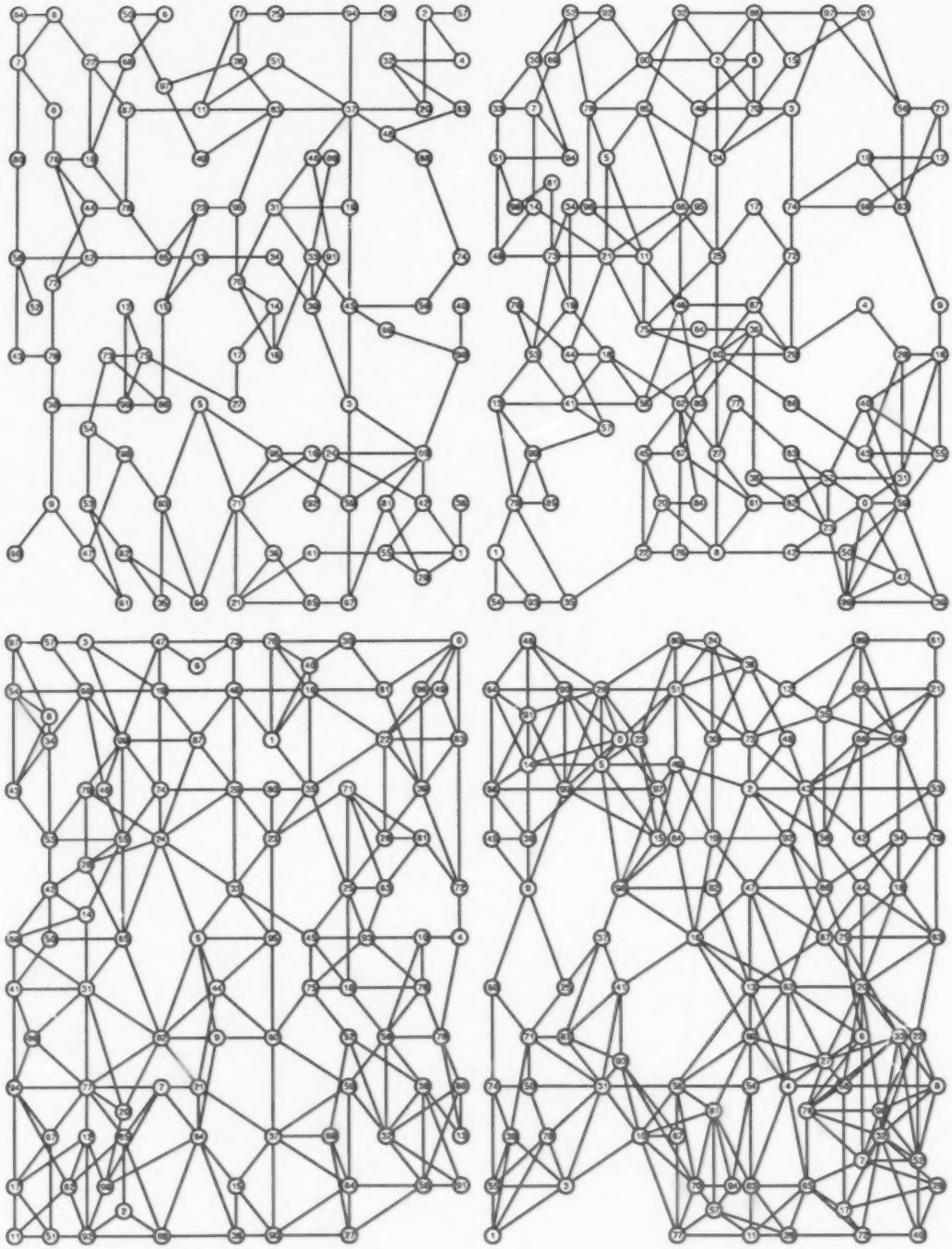


Figure B.9 100-Node Networks

Appendix C

ICH and SLPA Software User Documentation

C.1 Description of the SLPA Tool

The spare link placement subsystem is a mechanized engineering tool kit for evaluating various aspects of the restorability of a network. With this program, it is possible to measure restorability and over-restorability, manipulate sparing and working capacity, evaluate operation of various portions of the SLPA algorithm in isolation, and even initiate an automatic manipulation of the network using the SLPA algorithm.

The automatic spare capacity placement algorithm is novel in that it takes a classical NP-hard problem and approaches it by a heuristic method. The tool minimizes capacity while attempting to achieve the desired level of restorability. There is currently no way to specify anything other than 100 % target restorability.

It is possible to run the program in a path table mode, a method which speeds execution rate. See the description of the command line mode for further details.

This user manual provides a brief description of each of the user options available in the SLP program, a description of the command line options, and briefly describes the basic input and output files.

The executable version of the SLPA subsystem is started by typing **slp** at the UNIX prompt. (Ensure that the search path includes the directory containing the executable !) SLP can be started in 2 modes: Path Table and Iterated Dijkstra

Upon startup, the user is asked for the maximum restoration path length (RPL) to be used in the searches for restoration paths. This value, which is not range-checked, is a positive integer. Generally, RPL is less than or equal to 10, and an RPL value greater than the number of spans (S), is not valid. Next, the user is queried for the short-form network file. The actual network topology files at TRLabs exist in 2 main formats. The first is 'long' form, in which the node interconnect information and the relative node positioning information is provided. The second, newer, one is the 'short' form file, which does not contain the physical relative orientation information of the nodes. SLP uses short form files. The path for the file must be specified. Note that the program does not understand UNIX short-cut conventions in the path specification, and thus the explicit path or a relative path from the working directory must be specified.

C.1.1 Menu Options

Each of the options presented to the user upon starting SLP is described below.

Option 0: Edit Sparing Values

This option allows the user to incrementally modify the number of spare systems for a particular span or all spans in a network. The user may enter the span number (0..(S-1) {where S = number of spans}) or may enter -2 to globally modify all spans by the same incremental value. If a single span is entered, the current link total is displayed followed by a prompt for the

incremental change in systems. Any integer value may be entered, however, the program will not allow the number of spare links to drop below 0. As a result, a simple way to zero the spare capacity in the global spare link edit mode is to enter a very large negative integer for the incremental number of systems.

Option 1: List Sparing Values

Invoking option 1 generates a table displaying the sparing values for each node in the network under consideration. A sample output follows:

The spare values are:
(s 0= 2), (s 1= 4), (s 2= 4), (s 3= 5), (s 4= 1),
(s 5= 5), (s 6= 5), (s 7= 2), (s 8= 5), (s 9= 3),
(s 10= 4), (s 11= 1), (s 12= 1), (s 13= 3), (s 14= 2),
(s 15= 4), (s 16= 3), (s 17= 4), (s 18= 4), (s 19= 2),
(s 20= 5), (s 21= 3), (s 22= 5), (s 23= 5), (s 24= 5),
(s 25= 1), (s 26= 5), (s 27= 4), (s 28= 1), (s 29= 2),

In the format (s *m*= *n*), *m*= span number and *n*= number of spare links.

Option 2: Current Restorability

This option gives a summary of the span restorabilities, based on current network parameters. Below is a sample of option 2 output:

The New Span Restorabilities are:

(s 0= 6(/ 6)), (s 1= 5(/ 6)), (s 2= 3(/ 3)), (s 3= 2(/ 5)), (s 4= 3(/ 9)),
(s 5= 3(/ 10)), (s 6= 1(/ 3)), (s 7= 5(/ 7)), (s 8= 4(/ 7)), (s 9= 4(/ 9)),
(s 10= 1(/ 1)), (s 11= 7(/ 9)), (s 12= 2(/ 2)), (s 13= 5(/ 8)), (s 14= 5(/ 5)),
(s 15= 4(/ 6)), (s 16= 5(/ 5)), (s 17= 4(/ 8)), (s 18= 4(/ 10)), (s 19= 5(/ 9)),
(s 20= 6(/ 7)), (s 21= 4(/ 10)), (s 22= 3(/ 9)), (s 23= 4(/ 10)), (s 24= 5(/ 8)),
(s 25= 4(/ 7)), (s 26= 3(/ 10)), (s 27= 3(/ 3)), (s 28= 8(/ 9)), (s 29= 6(/ 6)),

Restorability = 124 (/207)

Redundancy = 0.48309

In the format (s i=k_i(/ w_i)), i=span number, k_i = number of restorable links, and w_i = number of working links.

Option 3: Evaluate add 1 fibrecap for each span

This option shows what would happen to the network under consideration if one spare transmission system, also known as module (a module being the number of links specified by the modularity figure for the system) or fibrecap, is added to a single span in the network. The results are shown as restorability in links on a per-span basis, since the single added spare system is tried at every span in the network to show the differential benefit of each potential placement. This option does not actually affect the network; it just provides a quick answer to a 'what if' question.

This option is intended primarily as an algorithm development aid.

Option 4: Evaluate sub 1 fibrecap for each span

This option shows what would happen to the network under consideration if one spare system is subtracted from a single span in the network. The results are shown as restorability in links on a per-span basis, since it is desired to show the relative effect of single spare system removal from every possible span. This option does not actually affect the capacities; it just provides a quick answer to a 'what if' question.

This option is intended primarily as an algorithm development aid.

Option 5: Change fibrecap

Change fibrecap allows the user to globally change the modularity of the transmission systems which comprise spans. This option does not perform a range check, so the user must be careful to enter integer values greater than 0. It is not possible to change the modularity of the network on a per-span basis. If the modularity is changed, all span capacities will be increased until an integral number of systems exist. Note that in this case, the added capacity is spare. Changes to modularity cannot be undone, because if modularity is incremented or decremented after such a change, the program is unable to determine what 'extra' sparing capacity was previously added to 'top up' the working system. Thus it is generally not possible to get back to the original sparing values, without re-reading the network file.

Option 6: Set Initial Sparing to 1

This option globally sets the sparing to 1 system if the modularity is 1. For modularity value n (where n != 1), s_i=fibrecap-(w_i modulus n) for each span i.

Option 7: Synthesize: iterate on add n

This option allows the user to execute the synthesize portion of the spare link placement algorithm. The user is prompted for n, the number of systems to add to spans. Note that there is no range checking, and n=0 causes program abends. After determining the value of n, the user is prompted for the range of examination. To determine the mode to use, the user must know whether the add1..addn-1have been performed to completion (failure to do further modifications). If they have not, then the full combinatorial search option ('0') must be used to give accurate results. This alternative can be very time-consuming as it is strictly order of complexity S^n (where S= number of spans, and n= number of links to be added). If they have, the n-only option ('1') would be used, as it increases solution speed by using shortcuts, relying on the fact that prior cases have failed.

This option is intended to be an algorithm development tool only, in conjunction with option 8, below.

Option 8: Tighten: Iterate on add n, sub n+1

Option 8 provides user access to the tightening phase of the spare link placement algorithm. It is used in conjunction with option 7 in an algorithm development mode. The user

performs one of the tightening phases specified in SLPA via this function. This process removes 'fully spare' systems; partially populated systems are not affected. After a value is supplied for 'n', the procedure proceeds to performing an addition of 'n' systems, followed by a subtraction of 'n+1' systems. This option does not perform range checking on the input provided to it. With this option, as with option 7, the user must know whether the add0_sub1...add(n-1)_subn tightening operations have been performed to completion (failure to do further modifications), to determine which mode to use. If these tightening functions have not, the full combinatorial search option must be used to give accurate results. If they have, the n-only option would be used, as it increases solution speed by eliminating evaluation of the candidates examined by the other search cases.

Option 9: Remove excess: iterate on sub1

This option attempts to remove single spare links that do not affect the restorability of the network.

Option 10: Auto synthesize and tighten

This option executes the automated full implementation of the spare link placement algorithm. The option executes until 100% or maximal non-100% restorability (in the case of a non-fully restorable network) is reached. Feedback is continuously provided as to restorability level and phase of execution. This is the option most likely to be used in a planning environment. This algorithm may be invoked immediately upon entering the subsystem, or after the network has been processed by other options. This option is usually executed after option 6.

Option 11: Erode: iterate

Option 11 removes spare systems recursively until no spare capacity remains. Further, the user is prompted for the type of erosion pattern desired, be it best case (least detriment to restorability per step), worst case (most impact to restorability per step), or random.

Option 12: Reread initial network (same topology)

This option allows the user to return to the saved network configuration.

Option 13: Change initial network (same topology)

Option 13 is similar to option 12, except that it prompts the user to enter the name of the network file to be retrieved. Note that a network with a different topology should not be read in at this point; if a different topology is to be studied, the user must exit this subsystem (using option 33) and re-execute it.

Option 14: Save current network

This option allows the user to save the network as it exists in the subsystem. The user is prompted for a filename.

Option 15: Change output files

This option has not been coded. It is meant to enable the user to change the output file in use.

Option 16: Add n spares to best locations

This feature will determine the best span or spans to which spare systems should be added. The user is prompted for the number of spare systems to add, and must also indicate whether a search is to be performed using short cuts based on restoration path length, or a full combinatorial search is to be performed.

With this option, as with option 7, to determine the mode to be used, the user must know whether the add1..addn-1 have been performed to completion (failure to do further modifications). If they have not, then the full combinatorial search option must be used to give accurate results. If they have, the n-only option would be used, as it increases solution speed by using shortcuts, relying on the fact that the previous cases have failed.

The option returns the network restorability if it is enhanced by this process; otherwise nothing is returned.

Option 17: Evaluate over-restorability of spans

This option allows the user to non-intrusively evaluate the current over-restorability of the spans in the network. In this context, over-restorability is defined as the number of additional working links which could be added to span i yet leave span i fully restorable. A sample output is provided :

The Span Over Restorabilities are:
(s 0= 1), (s 1= 0), (s 2= 0), (s 3= 0), (s 4= 0),
(s 5= 0), (s 6= 0), (s 7= 0), (s 8= 0), (s 9= 0),
(s 10= 3), (s 11= 0), (s 12= 5), (s 13= 0), (s 14= 0),
(s 15= 0), (s 16= 0), (s 17= 0), (s 18= 0), (s 19= 0),
(s 20= 0), (s 21= 0), (s 22= 0), (s 23= 0), (s 24= 0),
(s 25= 0), (s 26= 0), (s 27= 1), (s 28= 0), (s 29= 0),
The Actual Restorability is = (124/207)

The Maximal Restorability is = 134

In the format (s i= OR_i), i = span number, and OR_i= over-restorability for span i. Note that the actual restorability of the network is given as well as the network maximal restorability. The maximal restorability is calculated by summing the individual span over-restorabilities and adding the result to the current network restorability figure.

Option 18: Edit working values

Option 18 allows the user to incrementally modify the number of working links per span on a span by span basis, or globally (by entering -2 as the span number). For the former method, the current integer value is presented before the user is prompted for the incremental value. No

range checking is done on the incremental value entered. It is not possible to force the working values to be negative.

Option 19: Print pathtable

This option 'prints' the path-table to a file called 'pathtable.out'. This option only works when the program is in path-table mode.

Option 20: Add a path

Option 20 allows the user to add one restoration path (a path of spare links) to the network. It echoes restorability if a path was added; otherwise, it returns nothing.

Option 21: Dump Pathtable

This option may only be used when running in path table mode. The path table information is dumped into a file in compressed format. The user is prompted for the output file name. The file is subsequently loaded in a future slp invocation on the same network topology with the command line option [-l path table].

Option 22: Menu

This option provides a list of the available menu options.

Option 23: Grow:iterate

This option recursively grows a network by provisioning working links from the pool of spare capacity, thus decreasing network restorability. The user is prompted for a destination file name for the output data (grow.out is the default offered), as well as for the type of growth pattern desired: best case (minimizing restorability impact), worst case (maximizing restorability impact), or random. This option terminates execution when no spare capacity remains.

Option 25: Auto synthesize and tighten (fast)

This option functions similarly to option 10 (the automated complete implementation of the algorithm), except that it eliminates the add2_sub3 phase, in order to increase execution speed.

Option 26: Add a working path (eval)

Option 26 non-intrusively evaluates the effect on restorability of adding a working path to the network. A working path is a one link wide set of contiguous, non-looping links between two desired nodes. The user may supply the nodes between which the path is to be added, or the system can randomly determine (based on a user-supplied random number seed) the source-sink pair between which to add a path. The user may also supply the path information via a file. This file format consists of a pair of space-delimited integers in the range of 0..N-1, where N = number of nodes. This option does not work in path-table mode. The user is also prompted for the name of a destination file (default is: 'pair1.out'). Feedback provided includes the nodes which have been selected (in the random node addition case), as well as a listing of the best spans to use for the shortest path route. If it is not possible to give a complete route, a minimum impact solution is

suggested. Before implementation, the user is asked whether the change is to be accomplished by redesigning the whole network (single sparing pool mode initiated), or as an additional capacity change (dual sparing pool mode initiated). The user also has the option of scrubbing the whole change at this point. Option 26 does not actually perform a change; it just suggests what would happen if a change were done.

Option 27: Add a working path (if sparing available)

Option 27 adds a working path (using a single link between each of the nodes forming a shortest path) to the network if sufficient sparing capacity is available for use as the working path. Its operation is otherwise the same as option 26. This option does not work in path-table mode.

Option 28: Add a working path (add capacity if needed)

Option 28 adds a working path to the network, adding working capacity, in the form of 'spared' systems between nodes if no sparing is present. Its operation is otherwise the same as option 26. This option does not work in path-table mode.

Option 29: Return to single sparing pool

This option, used in conjunction with option 30, is used to set the network up with a single sparing pool.

Option 30: Change to two sparing pools: old and new

This option forces the network into a dual sparing pool mode. The resulting two sparing pools consist of a 'floor spare' and a 'spare' pool. The 'floorspare' pool consists of all of the spare capacity present in the original (single) sparing pool. The 'spare' pool will contain all of the incremental sparing added after this point. The purpose of this is to protect the previously existing sparing capacity from the upcoming network modifications. When option 29 reforms a single sparing pool, it simply sums the contents of both of these pools. The same functionality is accomplished 'on the fly' in options 26-28.

Option 31: Switch to over-restorability mode

This option, used in conjunction with option 32, changes the program operation from restorability mode to 'greater than 100 % target restorability' mode. This mode enables the user to explore the effects of adding excess capacity to a network.

Option 32: Switch to normal (restorability) mode

This option changes the operating mode from 'greater than 100 % target restorability' mode back to restorability mode. It is used in conjunction with option 31.

Option 33: Exit

Entering this option terminates execution of the program.

C.1.2 Command Line Options

The various command line options available are summarized below:

slp [[-i[o] networkfilename repeatlimit]
[-f fibrecap]
[-l pathtablefilename_in]
[-b[c][f] resultsfilename [-o outntwkfilename]
[-d pathtablefilename_out] [-s statsfilename]]
[-e resultsfilename erosiontype randomseed]
[-g resultsfilename growthtype randomseed]
[-t {type of calc_rest (0-dijkstra or 1-pathtable)}]
[-m {0-logical,1-actual,2-logical+actual}]
]

[[-i[o] networkfilename repeatlimit]

Entering this line during slp invocation allows the user to specify the name of the network file being used as well as the repeat limit desired. If option 'o' is entered following option 'i', then the network will not be built; rather the network (using path tables) is only initialized. The 'networkfilename' field is where the short form network file to be used is identified. The 'repeatlimit' field is where the maximum restoration path length is specified.

[-f fibrecap]

Entering '-f' enables the user to specify the global transmission system module size during invocation.

[-l pathtablefilename-in]

The '-l' option enables the user to specify the path table name for path table mode operation. This path-table must already exist from a previous slp run.

[-b[c][f] resultsfilename [-o outntwkfilename]
[-d pathtablefilename-out] [-s statsfilename]]

The '-b' option is the build option. This option is entered when it is desired to do a fully automated optimization of the spare capacity of the network. Entering '-b' is equivalent to invoking options 6,10, and 14. The 'c' option, when specified, ensures that the program will not start by setting the spare capacity to 1; rather, spare capacity present at program invocation time will be used. The 'f' option, when used with '-b' and possibly 'c', invokes a spare capacity placement algorithm equivalent to that used in option 25 ('Auto synthesize and tighten (fast)'). The 'resultsfilename' field, which names the sink for the results information, must be specified if option 'b' is used. It is optionally possible to specify the output network file name, using the -o option. If it is not specified, the user will be queried for it . It is also possible to use the '-s' option to specify the sink file for the statistics generated for the run.

[-e resultsfilename erosiontype randomseed]

This option invokes the erosion mode (option 11). The user must specify a file for the results, the erosion type ('0': best case,'1': worst case, '2': random). Further, a random number seed must be provided to determine the initial span impacted.

[-g resultsfilename growthtype randomseed]

This invokes option 23 upon program invocation. The results file must be specified, as must the growth type ('0': best case, '1': worst case, '2': random). A random number seed must be provided to determine the starting node.

[-t type of calc rest]

This option determines the type of restoration calculation to be performed. The user has the option of Dijkstra or path table methods. The path table method is faster, but has higher computer storage resource demands. Iterated Dijkstra is the default.

[-m k-sp-selection-type]

When Dijkstra is selected for restorability calculations, this option allows the path selection order to be logical ('0'), actual distance ('1'), or a combination of the two ('2') which uses logical length values until a choice is required between two equivalent logical length alternatives, when the actual lengths are examined. Logical is the default selection type because it executes fastest.

C.1.3 Input and Output Files

Figure C.1 presents the short network descriptor file format.

N					
S					
1	u_1	v_1	d_1	s_1	w_1
	.				
	.				
	.				
i	u_i	v_i	d_i	s_i	w_i
	.				
	.				
	.				
S	u_S	v_S	d_S	s_S	w_S

Figure C.1 The Short Network Descriptor File Format

In this file: N is the number of nodes. S is the number of spans. For span i, u_i, v_i, d_i, s_i, and w_i are the source of span i, the sink of span i, the number of spare links on span i and the number of working links on span i, respectively. For example, Figure C.2 shows the descriptor file for the 20-Node, 30-Span (d_{avg}=3) study network used in this work.

```
20
30
  1   0   3  1  4  8
  2   0  17  1  4  2
  3   0  14  1  2  9
  4   0   2  1  3 10
  5   1  10  1  2  9
  6   1  19  1  5  5
  7   1   7  1  3  1
  8   2   6  1  5  5
  9   2   8  1  1  7
 10   3  12  1  4 10
 11   3   6  1  3  4
 12   3  14  1  1  3
 13   4   5  1  4  3
 14   4   9  1  2  5
 15   5   9  1  4  6
 16   7  10  1  4  3
 17   8  10  1  3  7
 18   8  13  1  2  2
 19   9  18  1  5 10
 20   9  12  1  2  8
 21  10  13  1  5  9
 22  11  15  1  3  5
 23  11  14  1  2  7
 24  11  16  1  1  1
 25  12  15  1  5 10
 26  13  15  1  1  8
 27  13  17  1  3 10
 28  15  19  1  3 10
 29  16  17  1  1  8
 30  18  19  1  2  4
```

Figure C.2 The Short Network Descriptor File for the N=20, d_{avg}=3 Study Network

Figure C.3 presents the short network descriptor file format.

$Red_{n,1}$ $R_{n,1}$ $s_{0,1}$ $s_{1,1}$ $s_{2,1}$ ... $s_{S,1}$
.
.
.
$Red_{n,i}$ $R_{n,i}$ $s_{0,i}$ $s_{1,i}$ $s_{2,i}$ ... $s_{S,i}$
.
.
.

Figure C.3 The Output File Format

Each time that the network spare capacity is changed, a line is printed to this file. Each line lists the network conditions after the capacity change is implemented, including: the network redundancy, the network restorability, and a vector of spare link quantities for the spans of the network. For example, Figure C.4 contains the output file for the 20-Node, 30-Span (d_{avg}=3)

study network used in this work. Initially, all spans have a single spare link except for span 5 which had a single link added in that first operation. The operations all add spare capacity (FS phase of SLPA) until column two reaches 100%. As column one shows, the final 5 operations reduce the redundancy of the network (DT phase of SLPA).

34.83	53.93	1	1	1	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
35.96	56.18	1	1	1	1	1	2	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
37.08	57.30	1	2	1	1	1	2	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
38.20	59.55	1	2	1	1	1	2	1	1	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
39.33	60.67	1	2	1	1	1	2	2	1	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
40.45	61.80	1	2	1	1	1	2	3	1	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
41.57	62.92	1	2	1	1	1	2	3	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
42.70	64.04	1	2	1	1	1	2	3	3	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
43.82	65.17	1	2	1	1	1	2	3	3	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
44.94	66.29	1	2	1	1	1	2	3	3	2	2	3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
46.07	67.42	1	2	1	1	1	2	3	3	2	2	3	1	1	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1
47.19	68.54	1	2	1	1	1	2	3	3	2	2	3	1	1	1	1	3	1	1	1	1	1	1	1	1	1	1	1	1	1
48.31	69.66	1	2	1	1	1	2	3	3	2	2	3	1	1	1	1	3	1	1	1	1	1	1	2	1	1	1	1	1	1
49.44	70.79	1	2	1	1	1	2	3	3	2	2	3	1	1	1	1	3	1	1	1	1	1	1	2	1	2	1	1	1	1
50.56	71.91	1	2	1	1	1	2	3	3	2	2	3	1	1	1	1	3	1	1	1	1	1	1	2	1	2	1	2	1	1
51.69	73.03	1	2	1	1	1	2	3	3	2	2	3	1	1	1	1	3	2	1	1	1	1	1	2	1	2	1	2	1	1
52.81	74.16	1	2	1	1	1	2	3	3	2	2	3	1	1	1	1	3	2	1	1	1	1	1	2	1	3	1	2	1	1
53.93	75.28	1	2	1	1	1	2	3	3	2	2	3	1	1	1	1	3	2	1	1	1	1	1	2	1	3	1	2	2	1	1
55.06	76.40	1	2	1	1	1	2	3	3	2	3	3	1	1	1	1	3	2	1	1	1	1	1	2	1	3	1	2	2	1	1
57.30	77.53	2	2	1	1	1	2	3	3	2	3	3	1	1	1	1	3	2	1	1	1	1	1	2	1	3	1	3	2	1	1
58.43	78.65	2	2	1	1	1	3	3	3	2	3	3	1	1	1	1	3	2	1	1	1	1	1	2	1	3	1	3	2	1	1
59.55	79.78	2	2	1	1	1	3	4	3	2	3	3	1	1	1	1	3	2	1	1	1	1	1	2	1	3	1	3	2	1	1
61.80	80.90	2	2	1	2	1	3	4	3	2	3	4	1	1	1	1	3	2	1	1	1	1	1	2	1	3	1	3	2	1	1
64.04	82.02	2	2	1	3	1	3	4	3	2	3	5	1	1	1	1	3	2	1	1	1	1	1	2	1	3	1	3	2	1	1
66.29	83.15	2	2	1	3	1	3	4	3	2	4	5	1	1	1	1	3	2	1	1	1	1	2	2	1	3	1	3	2	1	1
68.54	84.27	2	2	1	3	1	3	4	3	2	4	5	1	2	2	1	3	2	1	1	1	1	2	2	1	3	1	3	2	1	1
69.66	86.52	2	2	1	3	1	3	4	3	2	4	5	1	2	2	2	3	2	1	1	1	1	2	2	1	3	1	3	2	1	1
71.91	87.64	2	2	1	3	1	3	4	3	2	4	5	1	3	3	2	3	2	1	1	1	1	2	2	1	3	1	3	2	1	1
73.03	88.76	2	2	1	3	1	3	4	3	2	4	5	1	3	3	3	3	2	1	1	1	1	2	2	1	3	1	3	2	1	1
75.28	89.89	2	2	1	3	1	3	4	3	2	4	5	1	4	4	3	3	2	1	1	1	1	2	2	1	3	1	3	2	1	1
76.40	91.01	2	2	1	3	1	3	4	3	2	4	5	1	4	4	4	3	2	1	1	1	1	2	2	1	3	1	3	2	1	1
78.65	92.13	2	2	1	3	1	3	4	3	2	4	5	1	4	4	4	3	2	1	2	2	1	2	2	1	3	1	3	2	1	1
79.78	94.38	2	2	1	3	1	3	4	3	2	4	5	1	4	4	4	3	2	1	2	2	1	2	2	1	3	1	3	2	1	2
80.90	95.51	2	2	1	3	1	3	4	3	2	4	5	1	4	4	4	3	2	1	2	2	2	2	2	1	3	1	3	2	1	2
83.15	96.63	2	2	1	3	1	3	4	3	2	4	5	1	4	4	4	3	2	1	2	3	2	2	2	1	3	1	3	2	1	3
85.39	97.75	2	2	1	3	1	3	4	3	2	4	5	1	4	4	4	3	2	1	2	4	2	2	2	1	3	1	3	2	1	4
87.64	98.88	2	2	1	3	1	3	4	3	2	4	5	1	4	4	4	3	2	1	2	5	2	2	2	1	3	1	3	2	1	5
92.13	100.00	2	2	1	3	2	3	4	3	2	4	5	1	4	4	4	3	2	1	2	5	3	2	2	1	3	2	3	3	1	5
91.01	100.00	1	2	1	3	2	3	4	3	2	4	5	1	4	4	4	3	2	1	2	5	3	2	2	1	3	2	3	3	1	5
89.89	100.00	0	2	1	3	2	3	4	3	2	4	5	1	4	4	4	3	2	1	2	5	3	2	2	1	3	2	3	3	1	5
88.76	100.00	0	2	1	3	2	3	4	3	2	4	5	1	4	4	4	3	2	1	2	5	3	1	2	1	3	2	3	3	1	5
87.64	100.00	1	2	1	3	2	3	4	3	2	3	5	1	4	4	4	3	2	1	2	5	3	1	2	1	2	2	3	3	1	5
85.39	100.00	1	2	1	2	2	2	4	3	3	3	5	1	4	4	4	3	3	0	2	5	2	1	2	1	2	2	3	3	1	5

Figure C.4 The Output File for FS and DT with the N=20, d_{avg}=3 Study Network

C.2 Description of ICH Tool

No user interface has been written for the ICH method of SCP. In this work, ICH is implemented as a UNIX script (batch) file. The script file uses compiled 'C' programs and Mathematica™ to perform the functions which effect the ICH heuristic.

Any options which are investigated here, besides those which can be accessed through the command line, were implemented by editing the **ich** script file. Therefore, the script file is provided below with comments describing each function call. The command line is as follows:

ich 0 input_network RPL number_of_iterations feedback_type

Where:

input_network is the short network descriptor file (as was input to SLPA);

RPL is the restoration path length limit, therefore ICH RPL is assumed unless RPL is set to N (the number of nodes);

number_of_iterations is the maximum number of iterations of the ICH program attempted before the heuristic is aborted; and,

feedback_type is the type of algorithm which is used to assess restorability (input "mf" for max-flow restoration and "ks" for k-shortest path restoration).

The UNIX script file is presented in Figure C.5.

```
# ICH UNIX Script File
# inputs to the first call to this routine are
#       "ich 0 ntwkfilename max_rest_path_len max_iterations feedback_type"
# and subsequently, the first two variables are iteration number and iteration number + 1
# NOTE: ICH must be executed from a machine which has Mathematica installed.

set fb_type = "mf"
if ($#argv >= 5 && $5 == "ks") then
        set fb_type = "ks"
endif

alias ich 'csh -f ~/cutsets/ich'
alias cs '~/cutsets/css'
alias mi '~/cutsets/mis'
alias sstrip '~/cutsets/sstripds'
alias conv '~/spareplace/sparcsource/conv'
alias time '/usr/bin/time'

@ num1 = $argv[1]

if ($num1 < 1) then
        echo "starting iteration 1"
        echo `date` >starttime
        cp "$argv[2]" .
        cp "$argv[2]" net1
```

Figure C.5(a) The ICH UNIX Script File

```
        # Generate the set of incident cutsets for each span of the network.
        cs -ns net1 -a 5 $argv[3] 0 > /dev/null
        mv scutset.out scutset.out1
        # Convert the cutsets to constraints for input to the LP
        mi > /dev/null << +
        net1
        scutset.out1
        math.in1
+
        # Execute the Linear Program
        (time /usr/local/bin/math1.2 < math.in1 > m.out1) >& mtime.1
        cp math.out math.out1
        @ num4 = $argv[4]

        # Recurse.
        ich 1 2 $argv[3-$#argv] >>& batch.out
        exit 2
endif

# Exit if the maximum allowed number of iterations of ICH has been reached.
@ num4 = $argv[4]
if ($num1 > $num4 - 1) then
        echo `date` >aborttime
        exit
endif

echo "starting iteration $argv[2]"

# Strip all text from the LP output list of spare capacities
sstrip > /dev/null << +
        math.out
        spares$argv[1]
+

# Integrate new spare capacity values into the network descriptor file.
conv > /dev/null << +
        net$argv[1]
        s
        spares$argv[1]
        net$argv[2]
        q
+

# Exit if there is no change in the SCP between iterations of ICH
diff net$argv[1] net$argv[2] >! difffile
set d = `\ls -l difffile`
if ($d[4] <= 1) then
        echo `date` >! nodifftime
        exit
endif
unset d
```

Figure C.5(b) The ICH UNIX Script File (continued)

```
# Generate a new cutset (to be used as an LP constraint) for each unrestorable span
if ($fb_type == "mf") then
        # replaced line  << /usr/local/skiena/COMBINATORICA/Combinatorica.m
        cat >! math2.in << +
        << /usr/local/Math2.0/Packages/DiscreteMath/Combinatorica.m
        << ~/math.lib/maxFlow.m
        flows = totalFlow[ "$cwd/net$argv[2]"]
        flows >> "$cwd/math.out"
        Quit
+
        (time /usr/local/bin/math1.2 < math2.in > /dev/null) >& "mtime.$argv[2]b"

        cs -ns "net$argv[2]" -a 7 $argv[3] math.out > /dev/null
else
        cs -ns "net$argv[2]" -a 6 $argv[3] > /dev/null
endif

# Exit if no new constraint was found (this suggests full restorability was obtained).
set d = `\ls -l scutset.out`
if ($d[4] <= 1) then
        echo `date` >endtime
        exit
endif
unset d

# Add the new cutsets to the previous list of cutsets.
cat "scutset.out$argv[1]" scutset.out > "scutset.out$argv[2]"

# Convert the cutsets to constraint inequalities for use by the LP.
mi > /dev/null << +
        net$argv[2]
        scutset.out$argv[2]
        math.in$argv[2]
+

# Execute the LP on the new constraint set.
(time /usr/local/bin/math1.2 < "math.in$argv[2]" > "m.out$argv[2]") >& "mtime.$argv[2]"
cp math.out "math.out$argv[2]"

@ num2 = $argv[2]
@ num2++

# Recurse
ich $argv[2] $num2 $argv[3-$#argv] >>& batch.out
```

Figure C.5(c) The ICH UNIX Script File (continued)